Verification of memory access for deep learning compilers

Tung D. Le　小川 愛理　井上 拓　石崎 一明
Swagath Venkataramani　小原 盛幹

The succeed of deep learning recently leads to the emergence of many deep-learning-specialized accelerators that are designed as a distributed system with shared hierarchical memory. On the software side, compilers must take into account this complexity and ensure that the generated assembly program is correct. In this paper, we tackle the problem of verifying memory accesses or data transfers in an assembly program generated by our deep learning compiler, and report some preliminary results. Given an assembly program generated by the compiler from a source program, and a correct assembly simulator, we first run the simulator to simulate the execution of the assembly program in order to generate a sequence of data transfers. We then verify the sequence of data transfers against a correct sequence that is generated directly from the source program. We propose a novel data structure, called address tensor, that can capture both information of input data from the source program and addresses in the assembly program. Address tensors are very convenient to correctly generate a sequence of data transfers from the source program.They are also flexible enough to adapt to new data transfer strategies in the compiler. Our verification has been using during the development of the compiler, which helps us quickly find bugs and validate generated assembly programs.

1 Introduction

Deep learning has been emerging as an efficient tool for solving many important problems in computer vision, speech recognition, and natural language processing. It has been applied to traditional fields like software engineering as well as programming languages. The recent succeed of deep learning mainly comes from two things: a lot of labeled data and a lot of convenient compute power such as GPUs. Thought GPUs are very fast for training deep neural networks, they were not originally designed for deep learning but high performance computing in general. This leads to the emergence of many deep-learning-specialized accelerators such as TPU [4], RAPID AI accelerator [2]. Such accelerators have much better performance with low power consumption.

Deep learning accelerators often have thousands of small dedicated compute units with software-controlled shared hierarchical memory, which helps data move around the units in the fastest way and helps maximize parallelism. Though such a hardware design significantly accelerates deep learning applications, its flexibility in memory management by software makes a burden of software stack, especially, a compiler, taking care of all data transfers among many units, e.g. how to partition data among units, when to transfer and where to send or store the data to. It is often the case that small units has limited capacity such instruction sets, and

* Verification of memory access for deep learning compilers.
This is an unrefereed paper. Copyrights belong to the Author(s).
Tung D. Le, Eri Ogawa, Hiroshi Inoue, Kazuaki Ishizaki, Moriyoshi Ohara, IBM 東京基礎研究所, IBM Research, Tokyo.
Swagath Venkataramani, IBM TJ Watson Research Center, Yorktown Heights, NY.

the compiler needs to do a lot of optimizations, which causes difficulty in verifying that the compiler correctly generated all of the data transfers.

In this paper we tackle the problem of verifying memory accesses or data transfers generated by a compiler in DEEPTOOLS [8]—a software stack for RAPID AI accelerator [2]. In particular, we propose a method to verify correctness of the order of chunks of data transferred among memory levels in the accelerator, under a consumption that the AI accelerator's hierarchical memory is software-controlled. We propose a novel data structure, called address tensor, that can capture both information of input data from a source program of the compiler and addresses in a generated assembly code. Address tensors are very convenient to generate a correct sequence of data transfers from the source program. They are also flexible enough to adapt to new data transfer strategies in the compiler. Our verification has been using during the development of the compiler, which helps us quickly find bugs and validate generated assembly programs.

The rest of the paper is organized as follows: Section 2 gives an overview of the target compiler as well as the software stack and RAPID AI accelerator. Section 3 describes our verification method, how it is positioned in the software stack. It also describes the design of address tensor in detail, and some preliminary results. Section 4 concludes the paper and discusses future works.

2 Target Compiler

This section presents the overview of a compiler [6] for RAPID AI accelerator developed by IBM Research [2]. Before discussing the compiler, we quickly give an overview of the architecture of the RAPID AI accelerator and the DEEPTOOLS software stack [8] in which compiler is one of its components.

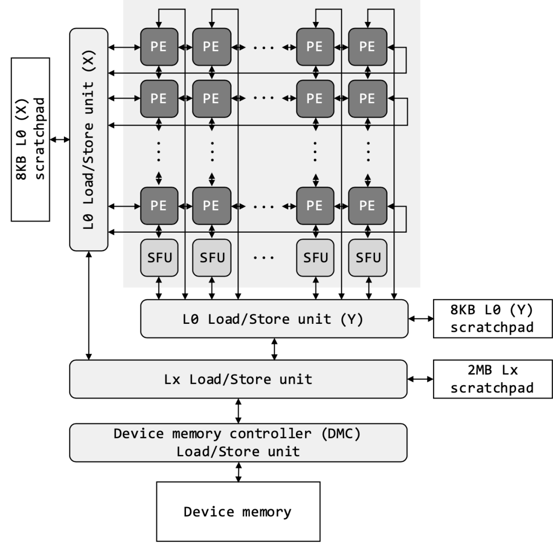

図 1 Architecture of the RAPID AI Accelerator

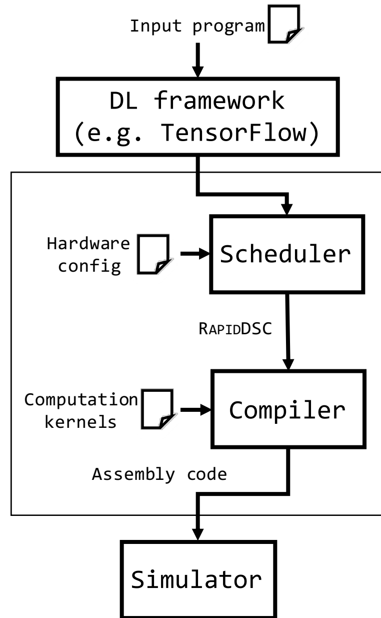

図 2 DEEPTOOLS Software Stack

2.1 Hardware and Software Architecture

2.1.1 Hardware Architecture

The RAPID AI accelerator [2], being developed by IBM Research, consists of an array of processing

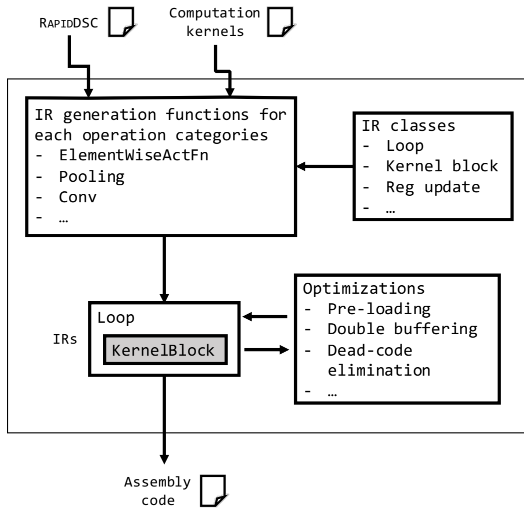

図 3 **Compiler Overview**

units (PEs) connected via nearest-neighbor FIFO links, an added row of special-function units (SFUs) and a 2-level scratchpad memory as shown in Fig. 1. The 8KB level-0 (L0) scratchpad is attached to both array dimensions, and connects to the 2MB next-level (Lx) scratpad. Each scratchpad has a pair of control units, i.e. a load unit (LU) and a store unit (SU). Data enters and leaves compute units (PEs and SFUs) through connections between the L0s and Lx scratchpads. All data transfers are software controlled, and the compiler must determine the load/store address for each data transfer instruction. All units have a carefully curated instruction set, a capacity-limited instruction buffer and a program counter, which is to achieve high-performance and energy-efficient deep learning processing. At a higher level, one processor chip consists of many cores that share an external device memory. The device memory controller (DMC) in a core is used to transfer data between the core's Lx scratchpad and the device memory.

2. 1. 2 DEEPTOOLS Software Stack

Figure 2 shows a simplified view of the DEEPTOOLS software stack developed for the RAPID AI accelerator [8]. We use the existing deep learning framework TensorFlow [1] as a front end. Users write their program, e.g. a deep neural network, in TensorFlow. TensorFlow then automatically generates a computational graph that is passed to a scheduler. The scheduler takes the computational graph generated by TensorFlow and the hardware parameters (e.g. the number of cores) as its input, and computes a mapping of the neural network onto the hardware architecture. In particular, it partitions the computations among the compute units and splits the data into multiple stages to be transferred to each level of the memory hierarchy. As a result, for each operator, there is a mapping, called RAPIDDSC, representing how data are split and transferred to each level of the memory hierarchy. The compiler finally generates optimized assembly code by using the RAPIDDSC and an annotated computation kernel for each operator (Sec. 2. 2). The assembly code can be run on an actual acceleration core or on a simulator.

2. 2 Compiler

Figure 3 shows the overview of our compiler. The compiler takes a RAPIDDSC and a computation kernel as inputs, and produces an assembly code for the accelerator.

Each operator, e.g. convolutional or fully-connected layer, is associated with a RAPIDDSC and a computation kernel template. The RAPIDDSC captures the spatial partitioning of the operator across the cores, loop ordering to complete the operator and data transfers through each unit [7].

In a RAPIDDSC, computation of an operator is described by using a 5-tuple $N_{(in,out,ij,mb,kij)}$ (Fig. 4), where 1) N_{in} is the number of input features, 2) N_{out} is the number of output features, 3) N_{ij} is the dimensions of each feature, 4) N_{mb} is the number of mini-batch inputs, and 5) N_{kij} is the

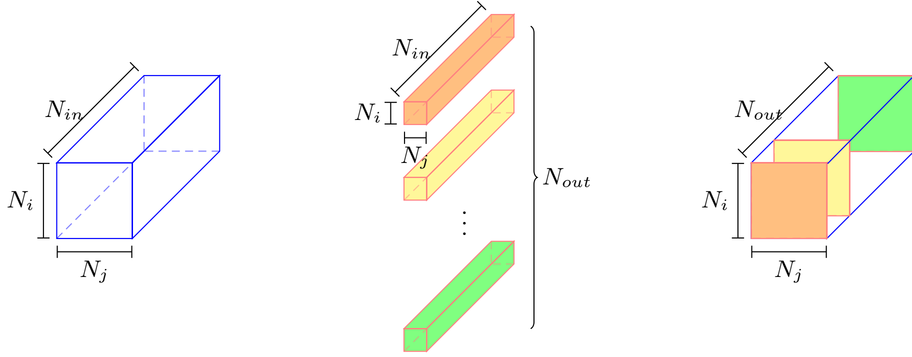

図 4 Data Layout for Input (left), Kernel (middle), and Output (right)

```
Workload
      Input: N(in, out, ij, mb, kij)
       Core: D(in, out, ij, mb, kij)
         Lx: B(in, out, ij, mb, kij)
         L0: T(in, out, ij, mb, kij)

Main loop: ["tpij", "tpmb", "tpkij", "tpin", "tpout", "btij", "btmb", "btkij",
            "btout", "btin", "dbmb", "dbkij", "dbout", "dbin", "dbij"]

Data transfer locations:
```

	Location	Data size	Source starting address	Destination starting address
d_{Inp}	dbkij	384	128	0
d_{Ker}	btin	9	512	1024

```
Dimension layout order:
         Inp: ["ij", "mb", "in"]
         Ker: ["ij", "in", "out"]
```

図 5 Example of a RAPIDDSC for an operator

dimensions of each kernel. The parameter N_{ij} is actually the product of two dimensions N_i and N_j. So is N_{kij}. From these parameters we can infer size of input, output or kernel data structure, in which, input feature size, Inp, is $N_{in} * N_{ij} * N_{mb}$ output feature size, Out, is $N_{out} * N_{ij} * N_{mb}$, and kernel size, Ker, is $N_{in} * N_{out} * N_{kij}$. Each data structure is associated with a layout dimension order that indicates how the data are stored in memory. A dimension layout order is represented by a list of dimensions, e.g. a layout dimension order for input is ["ij", "mb", "in"], where the first item in the list indicates the innermost partition that is stored in memory.

The scheduler partitions the computation to each core, i.e. $D_{(in,out,ij,kij,mb)}$ work allocated to each core, $B_{(in,out,ij,kij,mb)}$ data blocks reused from Lx, $T_{(in,out,ij,kij,mb)}$ data tiles reused from L0, and core $P_{(in,out,ij,kij,mb)}$ quantum of work fed to processing elements (PEs). It needs to explore a huge space of partitioning in order to find the best data partition strategy.

```
for _ in 1 to D_in/B_in
  for _ in 1 to D_out/B_out
    d_Inp
    for _ in 1 to D_ij/B_ij
      for _ in 1 to D_kij/B_kij
        d_Ker
        for _ in 1 to D_mb/B_mb
          b_Ker
          for _ in 1 to B_in/T_in
           for _ in 1 to B_out/T_out
             d_Out, b_Out
             for _ in 1 to B_kij/T_kij
               for _ in 1 to B_ij/T_ij
                 b_Inp
                 for _ in 1 to B_mb/T_mb
                   t_Ker, p_Ker, t_Out
                   for _ in 1 to T_out/P_out
                     for _ in 1 to T_in/P_in
                       t_Inp
                       for _ in 1 to T_kij/P_kij
                         for _ in 1 to T_mb/P_mb
                           for _ in 1 to T_ij/P_ij
                             p_Inp, p_Out
                             computation
```

図 6 Main loop of data transfer and computation

To complete work for a computation, each core needs to execute a main loop consisting of *15 nested loop* iterations clustered in 3 stages, i.e. (T/B), (B/T) and (D/B). The main loop is represented as a list of dimensions, e.g. [“tpij”, “tpmb”, “tpkij”, “tpin”, “tpout”, “btij”, “btmb”, “btkij”, “btout”, “btin”, “dbmb”, “dbkij”, “dbout”, “dbin”, “dbij”], where their first two characters denote memory levels, e.g. “db” means data between the external memory device to Lx scratchpad. In the main loop list, the first item is the innermost loop while the last item is the outermost loop. Value for, say, “dbmb” is computed by taking D_{mb} divided by B_{mb}. So are other dimensions’ values. It is the number of partitions obtained by splitting data along dimension “ij”, also denoting the number of iterations needed to transfer all of the partitions. To transfer data of input, output and kernel data structures between the different memory levels, there are 12 parameters indicating locations in the main loop to trigger the transfers, i.e $p_{(Inp,Out,Ker)}$, $t_{(Inp,Out,Ker)}$, $b_{(Inp,Out,Ker)}$, and $d_{(Inp,Out,Ker)}$. Fig. 6 shows an example of a main loop of data transfer and computation.

Note that, for each partition, we may send the whole partition at once, or divide the partition again into *chunks* and send chunks one-by-one, depending on the data transfer location for the data structure. For example, let’s consider a data transfer location, d_{Out} in Fig. 6 that is for output $D_{ij} * D_{mb} * D_{out}$, from device memory to Lx, and consider the dimension *out* of the output. Since d_{Out} is inside the loops db_{out} and bt_{out}, we do not send the whole partition of the output in device memory but we will divide the partition into chunks in which the dimension *out* of a chunk will be $D_{out}/db_{out}/bt_{out}$. So do the other dimensions.

Figure 5 shows an example of RAPIDDSC for an operator. We have information about workload such as input data, data for each core, Lx, and L0. We have a main loop that is executed at each core, and data transfer locations. For example, the location to transfer input from the external memory device to the core’s Lx scatchpad, d_{Inp}, is “btout” in the main loop, the size of input data is 384, and the input is stored at the address 128 in the external memory device and is sent to the address 0 in the Lx scratchpad. Finally, we have information about dimension layout order for each data structure.

The annotated computation kernel of an operator represents an operation template for the operator, including the operation category and kernel computation instructions for each hardware unit, hand-crafted by top-notch library developers.

Using the kernel instructions and the operation category from the kernel, as well as the RAPIDDSC from the scheduler, the compiler automatically generates outer loops for the kernel for each computation and data-transfer unit. For the DMC, the compiler generates its code to transfer data between the device memory and the Lx scratchpad. The library developers need to write instructions for all computation and data-transfer units except for the

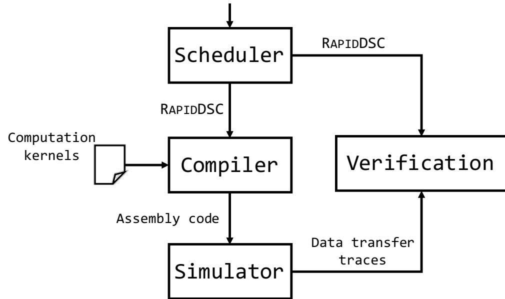

図 7 **Verification**

DMC only on one core. The compiler then automatically generates data transfer code for multiple cores by determining the actual address of data for each core.

3 Verification of Memory Access

Since the compiler needs to process a huge amount of work, e.g. generating code for every unit, transferring chunks of data to every compute and memory unit, and optimizing the generated code, it is non-trivial to ensure that the final generated assembly code is correct. A verification tool would be helpful for compiler developers to quickly verify the generated code [3] [5]. On the other hand, the tool should be flexible enough so that the developers can easily express their intention.

In this paper, we propose a method to help verify data transfers generated by the compiler. In particular, we verify whether data (input, output and kernel) are transferred between correct addresses or not. One data transfer means sending one datum (e.g. a number) from a memory address to another memory address. This paper only focuses on verify data transfers from the external memory device to each core, in particular, to the Lx scrathpad of a core. Verification of data transfers from Lx to L0 or to compute units is left for future work.

Fig. 7 shows how our verification is positioned in the software stack. It takes the output from the scheduler and the output from the simulator as inputs, and verifies the simulator's output against the description in the scheduler's output. The simulator can execute the whole assembly code and log information about each instruction. For verifying data transfer, we extract only traces about data transfer from the simulator's output.

The most important part in our verification framework is finding an intermediate representation (IR) that captures important information from both a RAPIDDSC and data transfer traces. In deep learning, it is common to use tensors (multiple dimension arrays) to represent data. Hence, we also use tensors as an IR. However, in our verification framework, values in a tensor are *memory addresses* instead of real values for input, output and kernel. We call such tensors *address tensor* (See Sec. 3. 2).

3. 1 Verification Method

Our purpose is to verify data transfers traced by the simulator, whether chunks of data are transferred to a memory level in *a correct order and to correct address or not*. To do so, we utilize address tensors to correctly generate a sequence of data transfers from a RAPIDDSC (output of the scheduler) (See Sec. 3. 3). Then, we verify the data transfers traced by the simulator against this sequence. Since data are split into chunks, it is meaningless to verify data transfers one by one. Hence, for each chunks of data, we generate a set of data transfers.

A sequence of data transfers traced by the simulator is represented by a list of tuples where each tuple represents a data transfer from a source to a destination. Let S be the sequence of data transfers traced by the simulator, its type is:

$$S :: [(\mathsf{src}, \mathsf{dest})]$$

where

$$\begin{aligned} \mathsf{src} &:: (\mathsf{Mem_Level}, \mathsf{Addr}) \\ \mathsf{dest} &:: (\mathsf{Mem_Level}, \mathsf{Addr}) \\ \mathsf{Mem_Level} &:: \mathsf{String} \\ \mathsf{Addr} &:: \mathsf{Integer} \end{aligned}$$

where [] denotes a list and () denotes a tuple.

A sequence of data transfers generated from a RAPIDDSC is represented by a list of *sets of tuples*, where each tuple represents a data transfer from a source to a destination. Let T be the sequence of data transfers generated from a RAPIDDSC, its type is:

$$T :: [\{(\mathsf{src}, \mathsf{dest})\}]$$

where {} denotes a set.

We propose two verifications: one is to verify the whole data transfers chunk by chunk, the other is to verify data transfers for each data structure. The reason to have the latter verification is that the compiler is able to do optimization for data transfers by doing overlap of data transfers and computation, in which each chunk is divided into sub-chunks and a sub-chunk is a triple of sub-chunks for input, output and kernel. Hence, if the former verification reports invalid data transfers, we need the latter verification to know for which data structure the invalid transfers happened.

3.1.1 Verifying Chunk Order

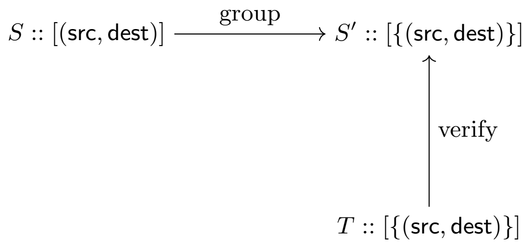

図 8 Verifying chunk order

Figure 8 shows how we verify a sequence of data transfers, S, traced by the simulator against the sequence T. We first create a new list, S', from S by grouping data transfers in S into sets so that the size of one set in S' is equal to the size of the corresponding set in T. Grouping is done from the first to the last element in S, so the size of the last set (or several last sets) in S' might be different from the corresponding one in T. After that we verify S' against T by comparing sets in S' and T one by one from the first to the last element. For each comparison, we use set equality, reporting 1) two sets are equal or not (data transfers exist in both sets), and 2) which data transfers in S' are missing in T and vice versa,

3.1.2 Verifying Chunk Order for Each Data Structure

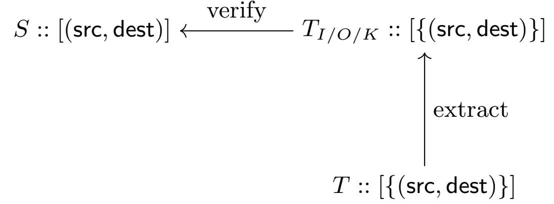

図 9 Verifying chunk order for a data structure

Figure 9 shows how we verify chunk order for a data structure. We first extract sequences of data transfers for each data structure, i.e. T_I for input, T_O for output, and T_K for kernel. Then, we verify S against each of the sequences.

However, S contains data transfers for all data structures. Hence, we need an algorithm to verify only data transfers of interest. Alg. 1 shows our algorithm to verify S again $T_{I/O/K}$. The key idea is to keep the maximum index in S after each chunk comparison. We start from the first element in S (Lines 1 and 5). For each chunk i $T_{I/O/K}$ (Lines 2–3), we find the existence of all elements of the chunk in S starting from max_idx (Lines 4–12). If a data transfer exists, we update max_idx to move forward in S. Otherwise, we report an invalid data transfer.

Input: $S, T_{I/O/K}$

1: $max_idx \leftarrow 0$
2: **for** $i \leftarrow 0, |T_{I/O/K}| - 1$ **do**
3: $chunk_i \leftarrow T_{I/O/K}[i]$
4: **for all** $dt \in chunk_i$ **do**
5: **if** $dt \in S[max_id:]$ **then**
6: $idx \leftarrow S.index(dt)$
7: **if** $idx > max_idx$ **then**
8: $max_idx \leftarrow idx$
9: **end if**
10: **else**
11: $Invalid$
12: **end if**
13: **end for**
14: **end for**

Algorithm 1: Verify data transfers for a data structure

3.2 Address Tensor

This section describes *address tensors* and basic operations to work with them. We implemented our verification framework in Python. Hence, we will use Python code to describe our address tensors.

Address tensor is a wrapper of a normal tensor in which we change its interface and add some additional operations so that it is more suitable to express RAPIDDSC.

To create an address tensor, we pass a layout and a starting memory address to its constructor,

```
class Tensor():
  def __init__(self, layout, mem_loc=None):
    ...
```

where layout is a list of tuples of dimension and item count, e.g. [("ij", 4), ("mb", 2), ("in", 2)]. The order of elements in the list follows the layout dimension order in a RAPIDDSC generated by the scheduler. An element in a tensor can be accessed by using operation where$(x, y, z, ...)$, where x, y, z etc. are dimension indices.

Figure 10 shows an example of an address tensor

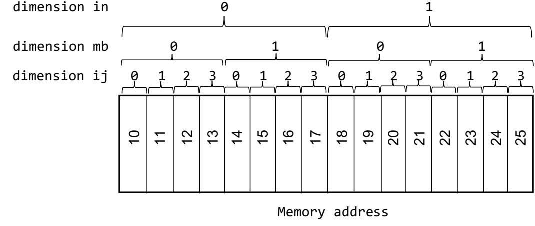

図 10 An example of an address tensor with layout [("ij", 4), ("mb", 2), ("in", 2)] and starting memory address 10

representing memory addresses for an input data structure that is stored in memory at address 10. To get the address of an item, say $in = 0, mb = 1, ij = 2$, calling where$(2, 1, 0)$ returns 16.

Actually, we use two different dimensions "i" and "j" (or "ki" and "kj") for "ij" (or "kij") to precisely capture the data structures. However, for brevity, we sometimes use "ij" or "kij" instead in this paper.

3.2.1 Basic Operations

Since data are partitioned and passed to compute and memory units, we provides operations for splitting an address tensor into multiple address tensors. A simple operation is splitting an address tensor along a specific dimension. Given a dimension and the number of partitions, the operation will split the tensor along such dimension into multiple address tensors. Its interface is

```
def split(self, dim, n_partitions=2)
```

To split a tensor along multiple dimensions, we provide operation split_n. Its interface is

```
def split_n(self, dims, flatten=False)
```

where dims is a list of tuples of dimension and number of partitions, e.g. [("ij", 1), ("mb", 2), ("in", 2)]. This operation will recursively call operation split to split the tensor, starting from the last item in the dimension list dims. It will return a nested dictionary of address tensors or a list of address tensors, depending on the value of the argument flatten. The nested dictionary is indexed

in the reversed order of dims for convenient use. In most cases, we would like to have a nested dictionary. In cases of biasadd operation and convolution operation with padding, we will flatten the dictionary and return a list of address tensors instead.

The last basic operation is padding an address tensor, which is used to present zero-padding for multiple dimension arrays. However, values used to pad the address tensor are not zeros but memory address. For this operation, we allow users to set a value as memory address for padding area in the address tensor, depending configurations in a RAPIDDSC. We will explain this design in detail when discussing verification of data transfers in Sec. 3. 1. The interface of this operation is

```
def pad(self, pad_width, value=-1)
```

where pad_width is a tuple of ((r_before, r_after), (c_before, c_after)) containing values to pad to the top/bottom row and leftmost/rightmost column, value is the value used to pad. This operation supports padding along only the two innermost dimensions. Hence, values "r" (row) and "c" (column) indicates "i" and "j" respectively, or "ki" and "kj" respectively, depending on whether data is input or kernel.

3. 2. 2 Utility Operations

As is often the case with RAPIDDSC, data are partitioned along multiple dimensions through *multiple loops*. Hence, we provide a special split operation that uses the loop information to partition an address tensor. Its interface is

```
def split_tensor_by_loop(ts, loop_dims)
```

where ts is the tensor we would like to split, loop_dims is a list of tuples of loop dimension and number of partitions, e.g. [("btij", 2), ("dbij", 1), ("dbmb", 2), ("dbin", 2)]. Note that, dimensions here are loop stages in the main loop in the RAPIDDSC. This operation will recursively call itself to split the input tensor along dimensions such as "ij", "mb" or "in", starting from the last item in the dimension list loop_dims. It will return a nested dictionary of address tensors. The nested dictionary is indexed in the reversed order of loop_dims for convenient use in the main loop.

Let us give an example about split_tensor_by_loop. Assume that we have an address tensor as the one in Fig. 10. If we split the tensor by using a list

$$[(\text{“}btij\text{”}, 2), (\text{“}dbmb\text{”}, 1), (\text{“}dbij\text{”}, 1), (\text{“}dbin\text{”}, 2)],$$

we will have a nested dictionary of address tensors in which each tensor has layout of

$$[(\text{“}ij\text{”}, 2), (\text{“}mb\text{”}, 2), (\text{“}in\text{”}, 1)].$$

We access these tensors by the indices in the nested dictionary, e.g. by using index (0, 0, 0, 1), we get the tensor at the first position of "dbin", first position of "dbmb", first position of "dbij" and second position of "btij".

Finally, we provide an operation to split an address tensor with overlapping, that is split_tensor_by_slicing. This is used in convolutional operator with padding. The idea is to split a tensor along one of its dimension into multiple partitions by slicing a window along that dimension with a stride. Its interface is

```
def split_tensor_by_slicing(ts, n_slices,
      stride=1, axis='r', flatten=False)
```

where ts is the tensor we would like to split, n_slices is the number of slices we would like to get, stride is the stride to move the window, axis is the dimension to split, flatten has the same meaning as the one in operation split_n. This operation supports splitting along only the two innermost dimensions. Hence, axis has only two values "r" (row) and "c" (column). If data is input, "r" and "c" correspond to "i" and "j", respectively. If data is kernel, "r" and "c" correspond to "ki" and "kj", respectively. Actually, slicing over the other dimensions does not make sense.

3. 3 Generating a Sequence of Data

Transfers

This section presents how to generate a sequence of data transfers from a RAPIDDSC by using address tensors. We take data structure input as example. The other data structures, output and kernel, follows the same method.

Assume that input's dimension layout order is ["ij", "mb", "in"], firstly, we create an address tensor, TS_N, for input with layout [("ij", N_{ij}), ("mb", N_{mb}), ("in", N_{in})] and memory location 0.

```
TS_D = TS_N = Tensor([("ij", N_ij),
                      ("mb", N_mb),
                      ("in", N_in)], 0)
```

TS_N is stored in the external memory device and it must be partitioned into multiple tensors for multiple cores. To do so, we just call operation split_n for TS_N.

```
TS_D = TS_N.split_n(dims=[("ij", N_ij/D_ij),
                          ("mb", N_mb/D_mb),
                          ("in", N_in/D_in)])
```

As a result, we have TS_D is a list of address tensors for each core. These tensors will be transferred to lower levels in a core such as Lx, L0, PE, which is done by the main loop.

Let us take the RAPIDDSC in Fig. 5 as example. Alg. 2 shows how to generate a sequence of data transfers, T, for input from an external memory device to Lx scratchpath. Before starting the main loop, we need to prepare tensors for transferring. For each location, we know the outer loop (the chunk loop) by looking at the location in the main loop. In this example, location is "dbkij", so the chunk loop according to the main loop must be [("dbout", x), ("dbin", y), ("dbij", z)] (Line 2), where x, y, z are the number of iterations for each loop calculated by taking $D_{out/in/ij}$ divided by $B_{out/in/ij}$, respectively. Using the chunk loop we split the address tensor for this core i, TS_D^i, into multiple tensors to transfer them to the lower memory level (Line 3). After that, we do the main loop. At the transfer location, e.g. "dbkij", we take the corresponding chunk based on the current iteration as a source chunk, create a destination chunk with the same layout at the lower memory level, transfer the source chunk to the destination chunk to obtain a set of data transfers, *seq*, and append the set *seq* to the return list T (Lines 7–9). Operation *do_transfer* just takes pairs of elements of the same index in *src_c* and *dest_c* and creates tuples of them, representing a data transfer from one memory address in the source memory device to one memory address in the destination memory device.

Input: TS_D^i (address tensor for core i)
Output: $T :: [\{(\mathsf{src}, \mathsf{dest})\}]$

```
 1: T ← []
 2: chunk_loop ← [("dbout", x), ("dbin", y), ("dbij", z)]
 3: chunk_map ←
        split_tensor_by_chunk(TS_D^i, chunk_loop)
    // Main loop
 4: for dbij ← 0, x − 1 do
 5:   for dbin ← 0, y − 1 do
 6:     for dbout ← 0, z − 1 do
 7:       src_c ← chunk_map(dbij, dbin, dbout)
 8:       dest_c ←
            Tensor(src_chunk.layout, dest_address)
 9:       seq ← do_transfer(src_c, dest_c)
10:       T ← T ++ [seq]
11:       for dbkij ← 0, w do
12:         ...
13:       end for
14:     end for
15:   end for
16: end for
```

Algorithm 2: Generating data transfers for input for a core

The algorithm 2 can be extended naturally to support pre-loading, double-buffering and sub-chunk transfers, which are optimization techniques in the compiler, by splitting a chunk (Line 7) again into sub-chunks and transferring the sub-chunks at a sub-chunk location that should be calculated in advance.

3.4 Preliminary Results

The verification method in this paper has been using during the development of our compiler for

```
===== Verify by chunks =====
There are 224 chunks/sub-chunks

[Verifying pre-loading ...]
The number of transfers: 225
passed.

[Verifying the sub-chunk 1 ...]
The number of transfers: 225
+++ {(('dm', '2163345'), ('lx', '448'))}
--- {(('dm', '2163345'), ('lx', '449'))}
failed.

[Verifying the sub-chunk 2 ...]
The number of transfers: 225
passed.

[Verifying the sub-chunk 3 ...]
The number of transfers: 225
+++ {(('dm', '2163345'), ('lx', '448'))}
--- {(('dm', '2163345'), ('lx', '449'))}
failed.

[Verifying the sub-chunk 4 ...]
The number of transfers: 225
passed.
```

図 11 Example of verifying BiasAdd operator in the VGG16 neural model

the deep learning accelerator core. It can automatically verify all operators in a neural network. During the development of deep learning operators in the compiler, the verification has been helping us identify bugs such as missing buffer switching in double-buffer transfers in BiasAdd operators in the VGG16 neural model, which is non-trivial to know if we look at an output from the simulator only (Fig. 11). Since address tensors used in the verification can capture both information of input data and addresses, they are very useful for compiler developers to check whether their address calculation for an operator is correct or not without manually calculating them by hand.

4 Conclusion

Developing a compiler for deep-learning-specialized accelerators is time-consuming and error-prone. In this paper, we propose a method to automatically verify data transfers generated by a compiler, which helps compiler developers detect and fix bugs quickly. Our key idea is proposing a data structure, address tensor, that can capture both multi-dimension information at a high level and address information at a low level. As shown in the paper, address tensor with a few operations helped us express the input of the compiler in a succinct way. It is also flexible enough for us to express different strategies implemented in the compiler.

In this paper, we borrowed a simulator to generate data transfers from a generated assembly code, where we assumed that the simulator behavior is correct. In the future, we would like to directly generate data transfers from the assembly code in a systematic way such as using an execution graph. Also, we will extend our method to verify data transfers at all memory levels in the accelerator.

参 考 文 献

[1] Abadi, M., Barham, P., Chen, J., Chen, Z., Davis, A., Dean, J., Devin, M., Ghemawat, S., Irving, G., Isard, M., Kudlur, M., Levenberg, J., Monga, R., Moore, S., Murray, D. G., Steiner, B., Tucker, P., Vasudevan, V., Warden, P., Wicke, M., Yu, Y., and Zheng, X.: TensorFlow: A System for Large-scale Machine Learning, *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation*, OSDI'16, Berkeley, CA, USA, USENIX Association, 2016, pp. 265–283.

[2] Fleischer, B., Shukla, S., Ziegler, M., Silberman, J., Oh, J., Srinivasan, V., Choi, J., Mueller, S., Agrawal, A., Babinsky, T., Cao, N., Chen, C., Chuang, P., Fox, T., Gristede, G., Guillorn, M., Haynie, H., Klaiber, M., Lee, D., Lo, S., Maier, G., Scheuermann, M., Venkataramani, S., Vezyrtzis, C., Wang, N., Yee, F., Zhou, C., Lu, P., Curran, B., Chang, L., and Gopalakrishnan, K.: A Scalable Multi-TeraOPS Deep Learning Processor Core for AI Training and Inference, *2018 IEEE Symposium on VLSI Circuits*, June 2018, pp. 35–36.

[3] Fox, A., Myreen, M. O., Tan, Y. K., and Kumar, R.: Verified compilation of CakeML to multiple machine-code targets, *Proceedings of the 6th ACM SIGPLAN Conference on Certified Programs and Proofs*, ACM, 2017, pp. 125–137.

[4] Jouppi, N. P., Young, C., Patil, N., Patterson, D., Agrawal, G., Bajwa, R., Bates, S., Bhatia, S., Boden, N., Borchers, A., Boyle, R., Cantin, P.-l., Chao,

C., Clark, C., Coriell, J., Daley, M., Dau, M., Dean, J., Gelb, B., Ghaemmaghami, T. V., Gottipati, R., Gulland, W., Hagmann, R., Ho, C. R., Hogberg, D., Hu, J., Hundt, R., Hurt, D., Ibarz, J., Jaffey, A., Jaworski, A., Kaplan, A., Khaitan, H., Killebrew, D., Koch, A., Kumar, N., Lacy, S., Laudon, J., Law, J., Le, D., Leary, C., Liu, Z., Lucke, K., Lundin, A., MacKean, G., Maggiore, A., Mahony, M., Miller, K., Nagarajan, R., Narayanaswami, R., Ni, R., Nix, K., Norrie, T., Omernick, M., Penukonda, N., Phelps, A., Ross, J., Ross, M., Salek, A., Samadiani, E., Severn, C., Sizikov, G., Snelham, M., Souter, J., Steinberg, D., Swing, A., Tan, M., Thorson, G., Tian, B., Toma, H., Tuttle, E., Vasudevan, V., Walter, R., Wang, W., Wilcox, E., and Yoon, D. H.: In-Datacenter Performance Analysis of a Tensor Processing Unit, *Proceedings of the 44th Annual International Symposium on Computer Architecture*, ISCA '17, New York, NY, USA, ACM, 2017, pp. 1–12.

[5] Myreen, Magnus O and Gordon, Michael JC and Slind, Konrad: Machine-code verification for multiple architectures-an application of decompilation into logic, *2008 Formal Methods in Computer-Aided Design*, IEEE, 2008, pp. 1–8.

[6] Ogawa, E., Ishizaki, K., Inoue, H., Venkataramani, S., Choi, J., Wang, W., Srinivasan, V., Ohara, M., and Gopalakrishnan, K.: A Compiler for Deep Neural Network Accelerators to Generate Optimized Code for a Wide Range of Data Parameters from a Hand-crafted Computation Kernel, *2019 IEEE Symposium in Low-Power and High-Speed Chips (COOL CHIPS)*, April 2019, pp. 1–3.

[7] Venkataramani, S., Choi, J., Srinivasan, V., Gopalakrishnan, K., and Chang, L.: POSTER: Design Space Exploration for Performance Optimization of Deep Neural Networks on Shared Memory Accelerators, *2017 26th International Conference on Parallel Architectures and Compilation Techniques (PACT)*, Sep. 2017, pp. 146–147.

[8] Venkataramani, S., Choi, J., Srinivasan, V., Wang, W., Zhang, J., Schaal, M., Serrano, M., Ishizaki, K., Inoue, H., Ogawa, E., Ohara, M., Chang, L., and k. gopalkrishnan: DeepTools: Compiler and Execution Runtime Extensions for RaPiD AI Accelerator, *IEEE Micro*, (2019), pp. 1–1.