Compiler-Assisted Operator Template Library for DNN Accelerators

Jiansong Li, Wei Cao, Xiao Dong, Guangli Li, Xueying Wang, Lei Liu, Xiaobing Feng

Compiler-assisted Operator Template Library for DNN Accelerators

Jiansong Li[1,2], Wei Cao[1], Xiao Dong[1,2], Guangli Li[1,2], Xueying Wang[1,2], Lei Liu[1](✉), and Xiaobing Feng[1,2](✉)

[1] Institute of Computing Technology, Chinese Academy of Sciences, Beijing, China
[2] University of Chinese Academy of Sciences, Beijing, China
{lijiansong,caowei,dongxiao,liguangli,wangxueying,liulei,fxb}@ict.ac.cn

Abstract. Despite many dedicated accelerators are gaining popularity for their performance and energy efficiency in the deep neural network (DNN) domain, high-level programming support for these accelerators remains thin. In contrast to existing researches targeting the whole DNNs, we choose to dive into details and review this problem from a finer-grained level, operators. Due to performance concerns, operator programmers may have to take hand-written assembly as their first choice, which is error-prone and involves many programming chores. To alleviate this problem, we propose TOpLib, a compiler-assisted template library. By providing a unified user-view abstraction, TOpLib allows programmers to express computational kernels with high-level tensor primitives, which will be automatically lowered into low-level intrinsic primitives via expression templates. Moreover, considering memory management is performance critical and the optimization strategy of expression template is limited to enumeration based rewriting rules, we implement TOpLib with a compiler-assisted approach. We address the memory reuse challenges into the compiler, which allows TOpLib to make full use of on-chip buffers and result in better performance. Experiments over 55 typical DNN operators demonstrate that TOpLib can generate scalable code with performance faster than or on par with hand-written assembly versions.

Keywords: DNN Accelerators · Template Library · Address Space Management

1 Introduction

In recent years, many dedicated DNN accelerators are gaining popularity for their energy and performance efficiency. They have been deployed in embedded devices, servers and datacenters [1, 12]. These accelerators focus on specific customization for the computations of DNNs. Typically, DNNs are usually expressed as computation graphs, where nodes represent basic operations (namely operators, e.g., convolution, pooling, activation), edges refer to data consumed or produced by these operators. These operators can be offloaded to accelerators to speed up computation. In this paper, we focus on the programming problems puzzling the underlying operator programmers. Due to performance concerns,

they may have to take hand-written assembly as their first programming choice. Coding a highly tuned operator kernel usually requires expert knowledge to manage every hardware detail, which involves a plethora of low-level programming chores. To illustrate, we use a simple feedforward MLP kernel with the sigmoid activation function as an example. The main computations are as follows: $y = \varphi(\sum_{i=1}^{n} w_i x_i + b) = \varphi(w^T x + b)$, where $\varphi(x) = e^x/(1 + e^x)$. Fig. 1a shows the implementation of feedforward MLP kernel at Cambricon-ACC [15] accelerator via hand-written assembly. For the sake of brevity, we omit the kernel explanation details. But for now it suffices to see that even this trivial MLP implementation involves many low-level programming chores. For example, the vector-vector and matrix-vector instructions, e.g., *VAV* and *MMV*, usually need special registers to store the memory address and the size of input data. Programmers have to manually allocate these registers and track the lifetime of each memory blocks, which is burdensome and error-prone. Besides, these CISC style instructions usually have special address alignment requirements, programmers have to manually check the address alignment of each memory block. This kind of low-level coding is very typical during the development of DNN operators.

To alleviate the low-level programming issues, we propose a compiler-assisted template library for operator programmers, namely TOpLib (short for **T**ensor **Op**erator **Lib**rary). TOpLib follows the philosophy of decoupling the programmers' view from the diversity of underlying hardwares. It provides an user-view abstraction for DNN accelerators. It uses C-like syntax as the surface language to represent abstract data types and operations. In terms of implementation, we can integrate the abstract data type and corresponding operations inside compiler or wrap a template library that optimizes at compile-time. The former may be straightforward and strong, but requires much engineering effort, posing a great challenge to compiler maintainers. The latter would be much more easily achieved by taking use of the meta-programming technique called expression template. But for the latter, the optimization strategy is limited to enumeration based rewriting rules. In this paper, we propose a hybrid approach to implement TOpLib. TOpLib relies on the meta-programming capability of the programming language provided by the DNN accelerators, e.g., expression template and operator overloading. Considering the memory management is performance critical, we use a compiler-assisted method to address the memory reuse challenges.

The rest of this paper proceeds as follows: Section 2 introduces the design principles of TOpLib. The implementation details are presented in Section 3. Section 4 describes experiment results. Section 5 and 6 discuss the related work and conclude respectively.

2 Design

2.1 User-View Abstraction

For most reconfigurable DNN accelerators, there are at least three levels of memory hierarchy: off-chip memory (DRAM), on-chip scratchpad memory (SPM) and registers. Unlike the caches of CPUs, which are managed by the hardware automatically and are invisible to programmers, the on-chip buffers of DNN accelerators are usually explicitly managed by programmers due to performance

```
// MLP kernel code
// $0: input size, $1: output size, $2: matrix size
// $3: input address, $4: weight address
// $5: bias address, $6: output address
// $7-$10: temp variable address

VLOAD $3, $0, #100       // load input tensor from address (100)
MLOAD $4, $2, #300       // load weight matrix from address (300)
MMV $7, $1, $4, $3, $0   // W*x
VAV $8, $1, $7, $5       // tmp=W*x+b
VEXP $9, $1, $8          // exp(tmp)
VAS $10, $1, $9, #1      // 1+exp(tmp)
VDV $6, $1, $9, $10      // y=exp(tmp)/(1+exp(tmp))
VSTORE $6, $1, #200      // store output tensor to address (200)
```

(a) MLP implementation via hand-written assembly at Cambricon-ACC.

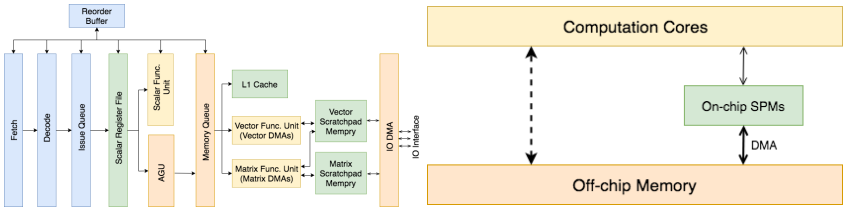
```
// MLP kernel code
// IN_SIZE: input size, OUT_SIZE: output size, MAT_SIZE: matrix size
// x: input tensor address, W: weight tensor address
// b: bias tensor address, y: output tensor address

__kernel__ void MLP(Tensor<DATA, half, IN_SIZE> x, Tensor<DATA, half, MAT_SIZE> W,
                    Tensor<DATA, half, OUT_SIZE> b, Tensor<DATA, half, OUT_SIZE> y) {
  Tensor<NEURAL, half, IN_SIZE> tx = load(x);   // load input data
  Tensor<NEURAL, half, OUT_SIZE> tb = load(b);  // load bias data
  Tensor<SYNAPSE, half, MAT_SIZE> tW = load(W); // load weight data
  Tensor<NEURAL, half, OUT_SIZE> ty = tW * tx;  // tmp = W*x
  ty += tb;                                     // tmp = W*x+b
  ty = exp(ty) / (1 + exp(ty));                 // y = exp(tmp)/(1+exp(tmp))
  store(ty, y);                                 // store result into off-chip DRAM
}
```

(b) MLP implementation via high-level tensor algorithmic primitives of TOpLib.

Fig. 1: MLP implementation at Cambricon-ACC via hand-written assembly and TOpLib's high-level tensor algorithmic primitives. For (a), note the ISA-level description with careful on-chip buffer layout design, manual register allocation and intentional memory address alignment checking.

and power concerns. To simplify the programmability of DNN accelerators, we hide the hardware execution details and expose the performance critical parts to programmers.

Fig. 2b shows View-ACC, an user-view abstraction for DNN accelerators. View-ACC consists of computation cores and on-chip SPMs, connected with wide buses to off-chip memory. Intuitively, the computation cores are responsible for the execution of DNN primitive operations, which will be thoroughly discussed in Section 2.3. The on-chip SPMs are abstractions for the on-chip buffers of DNN accelerators. They are fast but with size limitation, and are visible to operator programmers and compiler. They usually play the role of caching partial input data and temporary computational results. While the off-chip DRAMs are large but slow. They are used for the storage of large quantity DNN parameters and input/output data required by operators. The communications between the on-chip SPMs and the off-chip DRAMs are accomplished through explicit DMA load and store instructions.



(a) Prototype architecture of Cambricon-ACC [15].

(b) View-ACC: user-view abstraction of DNN accelerators.

Fig. 2: The architectural details of Cambricon-ACC and its user-view abstraction. The dashed arrow in (b) means there is a potential data-path between computation cores and off-chip DRAM.

Let's take Cambricon-ACC (Fig. 2a) as an example. Cambricon-ACC is a prototype DNN accelerator, which is based on the Cambricon DNN ISA [15]. In Cambricon-ACC, the off-chip DRAM is used for input and output data of DNN workloads. While the on-chip SPMs for vector and matrix function units, which are used for caching of neural and synapse data, can be treated as the on-chip SPMs of View-ACC. The scalar, vector and matrix function unit in Cambricon-

ACC can be treated as the computation cores of View-ACC. Other architectural components in Cambricon-ACC, e.g., reorder buffer, issue queue, fetch and decode components can be regarded as hardware implementation details, which are invisible to programmers. View-ACC hides the hardware execution details and exposes the performance critical parts to the programmers. Programmers can be released from the inner hardware execution details.

2.2 Memory Abstraction

Well-organized programs usually make frequent use of structs or classes. Array of Structures (AoS) and Structure of Arrays (SoA) describe the common techniques for organizing multiple structs/objects in memory [21]. In AoS, all fields of a struct are stored together. In SoA, all values of a fields of a struct are stored together. By our previous engineering efforts, we find that DNN operators are usually implemented in the SoA fashion. SoA benefits from the on-chip buffer utilization and is SIMD-friendly. However, in the DNN domain, the data participating in computation is usually high dimensional, not a single flat array. Therefore, we need a high-level type system with tensor as a first-class type. Tensor is a mathematical concept, which generalizes vector and matrix to higher ranks.

TOpLib represents tensor as the first-class type with *address space specifier*, an element type and shapes. Taking the memory hierarchy of DNN accelerators into consideration, TOpLib represents the hardware memory types by associating data with address spaces. All data will be expressed by aggregate type $Tensor$. Its definition will be like this, $Tensor < A, T, d_0, d_1, ..., d_{n-1} >$, where A denotes the address space of data. For Cambricon-ACC, its value can be NEURAL, SYNAPSE or DATA, which denotes the on-chip vector SPM, matrix SPM and off-chip DRAM respectively; T describes the data type, its value can be int, $float$, $half$ (16-bit signed floats) and quantized non-floating-point values such as currently supported $i8$ (8-bit signed integers); d_i represents the i-th dimension of the tensor. For a static tensor, the value of each dimension must be a compile-time known integer. For example, a single 32 by 32 image with 3 channels of color values between 0 and 255 stored in the off-chip DRAM of Cambricon-ACC, could be represented by $Tensor < DATA, i8, 3, 32, 32 >$.

2.3 Computation Abstraction

Table 1 lists up the most relevant tensor primitives for typical DNN workloads. To decouple the user-view from the diversity of underlying hardwares, TOpLib defines two-level tensor primitives. Programmers express operators with high-level tensor algorithmic primitives. And these algorithmic primitives will be lowerred into low-level intrinsic primitives. The low-level intrinsic primitives are listed at the bottom of Table 1. They are abstractions for the ISAs of DNN accelerators, which present as compiler builtin functions.

High-level Algorithmic Primitives In Table 1, we write $[T]_S^A$ for a tensor of data type T with a shape S at the address space A. Primitives *map* and *zip* are element-wise computational patterns. In the DNN domain, typical activation and normalization operators can be expressed with *map* and *zip* primitives. For

Table 1: High-level algorithmic primitives and low-level instrinsic primitives.

High-level algorithmic primitives	
$[T]^A_S = map(f, [T]^A_S)$	Apply f to each element of given tensor.
$[T]^A_S = zip(f, [T]^A_S, [T]^A_S)$	Apply f to each pair of corresponding elements of given two tensors.
$[T]^A_{S_1} = reduce(\oplus, [T]^A_{S_0})$	Collapse given tensor with the associative binary operator $\oplus$.
$[T]^{A_o}_{S_o} = conv([T]^{A_v}_{S_v}, \alpha_1, [T]^{A_h}_{S_h}, \alpha_2)$	Convolve input tensor v with range α_1 and a polynomial filter tensor h with range α_2.
$[T]^A_{S_1} = pool([T]^A_{S_0}, \alpha, \oplus)$	Combine adjacent elements of given tensor at region α by a reduction operation $\oplus$.
$[T]^A_{S_c} = matmul([T]^A_{S_a}, [T]^A_{S_b})$	Multiply two input 2-order tensors, i.e., matrices.
$[T]^{A_{dst}}_{S_{dst}} = load([T]^{A_{src}}_{S_{src}}, size)$	Load $size$ bytes of data from source tensor src.
$store([T]^{A_{src}}_{S_{src}}, [T]^{A_{dst}}_{S_{dst}}, size)$	Store $size$ bytes of data from source tensor src into destination tensor dst.
Low-level intrinsic primitives	
$\_\_map(f, p_1, p_0, n)$	Apply f to n elements starting at memory position p_0 and store results into p_1.
$\_\_zip(f, p_2, p_0, p_1, n)$	Apply f to binary element-wise pair of n elements starting from p_0 and p_1 and store results into p_2.
$\_\_reduce(\oplus, p_1, p_0, n)$	Collapse n elements starting from p_0 with associative binary operator $\oplus$ and store result to p_1.
$\_\_conv(p_o, n_o, p_v, n_v, \alpha_1, p_h, n_h, \alpha_2)$	Convolution: $v_{\alpha_1} \circledast h^{\alpha_2} \rightarrow o$.
$\_\_pool(p_1, n_1, p_0, n_0, \alpha, \oplus)$	Pooling n_0 elements starting from p_0 at region α by a reduction operation $\oplus$ and store the n_1 results to p_1.
$\_\_matmul(p_a, n_a, p_b, n_b, p_c, n_c)$	Matrix multiplication: $a \otimes b \rightarrow c$.
$\_\_load(p_{dst}, p_{src}, size)$	Load: $p_{dst} \xleftarrow{\text{size}} p_{src}$.
$\_\_store(p_{src}, p_{dst}, size)$	Store: $p_{src} \xrightarrow{\text{size}} p_{dst}$.

the *reduce* primitive, its operator $\oplus$ can be min, max or sum. Primitive *conv* denotes convolution operation, which can be expressed with $v_{\alpha_1} \circledast h^{\alpha_2}$. This expression convolves an input image v with range α_1 and a filter kernel h with range α_2. Primitive *pool* is a pooling operation. Its reduce operator $\oplus$ can be max, average or min. The primitive *matmul* represents for the matrix multiplication operation, which is usually used to express the inner product operations of fully connected layers in DNNs. While primitives *load* and *store* are abstractions of the data movement operations between the on-chip SPMs and off-chip DRAMs.

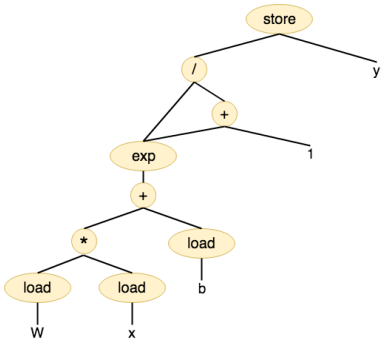
Low-level Intrinsic Primitives DNN accelerators usually provide SIMD instructions to accelerate computations. These SIMD instructions usually have the requirements of specific memory alignment and data layout. These low-level intrinsic primitives present as compiler builtin functions (see Table 1 below). Intrinsic primitives *__map* and *__zip* represent for element-wise instructions, such as the ReLU activation and vector addition. The reduce intrinsic functions consist of reduce min, max, sum. For Cambricon-ACC, the convolution, pooling and matrix multiplication intrinsic primitives have specific data layout requirements. The data layout of their input filters must follow the *NHWC* (N: batch size, H: height, W: width, C: channel) layout requirement. Primitives *__load* and *__store* denote the DMA communications between the on-chip SPMs and off-chip DRAM.

To demonstrate the representativeness of these high-level tensor primitives, we select several common DNNs running with ImageNet [5] dataset and typical ML workloads. We decompose their CPU execution time (Table 2). Obviously, these high-level algorithmic primitives characterize the DNN workloads mainly.

Table 2: Breakdown of execution time for each high-level algorithmic primitives in common DNN workloads. Note that primitive *load* and *store* is an abstraction for the data movement between off-chip DRAMs and on-chip SPMs, we omit them for CPUs.

Workloads	*map*	*zip*	*reduce*	*conv*	*pool*	*matmul*
AlexNet [13]	13.26%	12.07%	-	40.50%	3.39%	30.76%
GoogLeNet [22]	19.44%	17.30%	-	37.16%	25.86%	0.24%
Inception-V3 [23]	9.75%	6.65%	-	69.21%	14.08%	0.31%
MobileNetV1 [10]	21.05%	20.28%	-	58.37%	0.28%	-
ResNet50 [8]	18.06%	21.21%	-	58.55%	1.83%	0.35%
SqueezeNet [11]	11.27%	3.45%	-	63.37%	21.88%	-
VGG16 [20]	6.02%	-	-	75.77%	5.26%	12.94%
K-NN [3]	-	-	99.73%	-	-	-
SVM [9]	0.27%	0.14%	99.34%	-	-	-

Fig. 1b shows the feedforward MLP operator kernel written by high-level algorithmic primitives of TOpLib. In this case, binary arithmetic symbols, e.g., plus and slash are syntactic sugars for element-wise addition and division *zip* primitives. Compared with hand-written assembly version in Fig. 1a, MLP operator written by high-level algorithmic primitives are more intuitive to understand.



(a) Expression tree of MLP kernel.

```
// MLP kernel code
// IN_SIZE: input size, BIAS_SIZE:bias size,
// OUT_SIZE: output size, MAT_SIZE: matrix size
// px: input tensor address, pW: weight tensor address
// pb: bias tensor address, py: output tensor address

__kernel__ void MLP(half *px, int IN_SIZE, half *pW, int MAT_SIZE,
                    half *pb, int BIAS_SIZE, half *py, int OUT_SIZE) {
  __neural__ half x[IN_SIZE];            // tmp onchip buffer for input
  __neural__ half y[OUT_SIZE];           // tmp onchip buffer for output
  __neural__ half b[OUT_SIZE];           // tmp onchip buffer for bias
  __neural__ half t[OUT_SIZE];           // tmp onchip buffer
  __synapse__ half W[MAT_SIZE];          // tmp onchip buffer for weight
  __load(x, px, IN_SIZE * sizeof(half));   // load input
  __load(b, pb, OUT_SIZE * sizeof(half));  // load bias
  __load(W, pW, MAT_SIZE * sizeof(half));  // load weight
  __matmul(x, IN_SIZE, W, MAT_SIZE, y, OUT_SIZE); // y = W * x
  __tensor_add(y, b, y, BIAS_SIZE);        // y = y + b
  __exp(y, y, OUT_SIZE);                   // y = exp(tmp)
  __scale_add(1, y, t, OUT_SIZE);          // t = 1 + y
  __tensor_div(y, t, y, OUT_SIZE);         // y = y / t
  __store(y, py, OUT_SIZE * sizeof(half)); // store results
}
```

(b) Low-level instrinsic functions after lowering from high-level algorithmic primitives.

Fig. 3: Tensor expression tree of the feedforward MLP operator and the low-level instrinsic functions. In (b), *__tensor_add* and *__tensor_div* are the low-level *__zip* primitives; *__exp* and *__scale_add* are the low-level *__map* primitives.

3 Implementation

There are two ways to implement the tensor type system, i.e., integrate tensor type and the corresponding operations inside compiler or wrap a template library that optimizes at compile-time. The former is straightforward and strong, but requires much engineering effort. The latter would be much more easily achieved by taking use of the meta-programming technique called expression template. But for the latter, the optimization strategy is limited to enumeration based rewriting rules. Due to performance concerns, we implement TOpLib with a hybrid approach.

3.1 Expression Template

We use expression template to implement the mappings between high-level algorithmic primitives and low-level intrinsic primitives. Expression template is a

tricky implementation technique that uses the static evaluation abilities of compilers together with templates to construct a static expression tree at compile time [19, 24]. Fig. 3a shows expression tree of the feedforward MLP kernel. The trick of expression template is done by letting operations and functions return abstracted operation objects that contain all necessary information to construct the expression tree, instead of calculating the result themselves. Through the use of template meta-programming, it is even possible to manipulate the expression tree at compile time to apply algebraic transformations (enumeration based rewriting rules). For example, we define *fst* as a notational shorthand for $map(fst, x)$, where x is a n-order tensor, marked as $[x_0, x_1, ..., x_{n-1}]$, i.e., an array of tuples. Function *fst* yields a tensor which is composed of the first component of each tuple in x, i.e., $fst(x) = [x_0^{(0)}, x_1^{(0)}, ..., x_{n-1}^{(0)}]$. Programmers wrote such an algorithmic expression $y = fst(reduce(+_0, x))$①, where reduce operator $+_0$ means apply a reduction summation along the *first* dimension of the input tensor x. TOpLib will transform this expression into $y = reduce(+_0, fst(x))$②. Obviously, *reduce* operation $+_0$ is more computation-intensive than the *map* operation *fst*. Compared with expression ①, the *fst* operation in ② can filter out input data of the reduce operation and maintain the original semantics. This transformation can eliminate redundant computations, and thereby reducing the overall cost.

3.2 Compiler-assisted Optimizations

Considering the optimization strategy of expression template is limited to enumeration based rewriting rules, we conduct some compiler-assisted optimizations to guarantee performance.

Algorithm 1: Address Space Inference Optimization

```
Input  : 𝒦, kernel program of current DNN operator;
Output: ℳ, a map of pointers in a specific address space;
for Each kind of address space 𝒜𝒮_i; do
    𝒢𝒮 = ∅; // Collect all pointers guaranteed in address space 𝒜𝒮_i
    for Each pointer 𝒫 used in kernel 𝒦; do
        if 𝒫 is guaranteed to point to 𝒜𝒮_i; then
            𝒢𝒮.insert(𝒫);
    𝒮 = ∅; // Assume that all derived pointers point to 𝒜𝒮_i
    for Each instruction ℐ in 𝒦 returning a pointer type; do
        if ℐ is derived from other pointers; then
            𝒮.insert(ℐ);
    // Iteratively prove that they are in address spaces other than 𝒜𝒮_i
    bool changed = true;
    while changed do
        changed = false;
        for Each instruction ℐ in 𝒦 that is GEP, bitcast or PHINode; do
            for Each source Src of instruction ℐ; do
                if Src not in 𝒢𝒮 and Src not in 𝒮; then
                    𝒮.remove(ℐ);
                    changed = true;
    ℳ[𝒜𝒮_i] = 𝒮 ∪ 𝒢𝒮;
return ℳ;
```

Memory Address Space Inference As shown in Fig. 3b, the memory address space type qualifiers only apply to variable declarations, so compiler must infer the address space of a pointer derived from a variable. Besides, knowing the address space of memory accessing allows to emit faster load and store instructions. For example, a load from on-chip SPM is usually faster than the load from off-chip memory. We address this challenge using a compiler-assisted optimization pass (Algorithm 1). We implemente the address space inference through a fixed-point data-flow analysis [16]. The compiler runs propagate on $\mathcal{K}$ for each address space $\mathcal{AS}_i$. It first assumes all derived pointers (via pointer arithmetic) point to $\mathcal{AS}_i$. Then, it iteratively reverts that assumption for pointers derived from another one that is not guaranteed in $\mathcal{AS}_i$. finally, $\mathcal{GS}$ and $\mathcal{S}$ combined contains all pointers in memory space $\mathcal{AS}_i$.

Algorithm 2: On-Chip Memory Reuse and Off-Chip Data Promotion

Input : $\mathcal{K}$, kernel program of current DNN operator;
Output: $\mathcal{P}$, a set of memory partitions to be created;
$\mathcal{SB} = \emptyset$;// $\mathcal{SB}$ is a set of tensor memory blocks in $\mathcal{K}$
for *Each variable* $\mathcal{V}$ *in* $\mathcal{K}$; **do**
  if *GetDataType*($\mathcal{V}$) *is aggregate type* **then**
    Get address space and size of $\mathcal{V}$;// record meta-data of current block
    $\mathcal{SB}$.*insert*($\mathcal{V}$);
    Get live ranges of $\mathcal{V}$ *by data flow analysis*;
for *Each kind of address space* $\mathcal{AS}_i$; **do**
  Build interference graph $\mathcal{IG}$($\mathcal{V}$, $\mathcal{E}$), *where:*
  - $\mathcal{V}$ = { $\mathcal{B}_{i'} \in \mathcal{SB}$ | *GetAddressSpace*($\mathcal{B}_{i'}$)=$\mathcal{AS}_i$};
  - $\mathcal{E}$ *is an undirected edge connecting two memory blocks* ($\mathcal{B}_s, \mathcal{B}_t$), *if live ranges of* $\mathcal{B}_s$ *and* $\mathcal{B}_t$ *overlap*;
  Coloring $\mathcal{IG}$ *with greedy strategy*;
  for *vertices in* $\mathcal{IG}.\mathcal{V}$ *with the same color;* **do**
    Choose maximum size of colored memory blocks as current partition size;
  Reallocate partitions $\mathcal{P}$ *for memory blocks*;
  if $\mathcal{V}$.*size() !=* $\mathcal{P}$.*size()* **then**
    // reuse happens
    Promote partial off-chip data into remaining on-chip memory;
return $\mathcal{P}$;

Memory Allocator and Reuse Optimization Consider the motivation example in Fig. 4a, where memory blocks $B1$-$B5$ need to be allocated to a certain memory region. Assume the on-chip SPMs capacity of the DNN accelerator is $256K$. If compiler takes a naive linear allocator for these memory blocks, i.e., map each block to distinct memory locations (Fig. 4b), they will exceed the total capacity of on-chip SPMs. A careful inspection of the original operator's implementation reveals that memory block $B2, B3$ and $B5$ can in fact be shared, leading to the allocation in Fig. 4d. We can automatically achieves this goal by compiler static analysis without modifying the original kernel's implementation. Our memory reuse algorithm (Algorithm 2) is partially inspired by [14]. Firstly, the compiler collects the meta-data information (including address space and size) of tensor variables by statically walking through $\mathcal{K}$. Then It gets the live ranges of each tensor variable by data flow analysis. In this paper, we apply the definition of liveness for arrays in [14] to the aggregate tensor data type. Similarly, liveness analysis for tensors is conducted on the control flow graph (CFG) of $\mathcal{K}$. The liveness information for a tensor $\mathcal{T}$ can be computed on CFG of $\mathcal{K}$ by applying the standard data-flow equations to the entry and exit of every basic

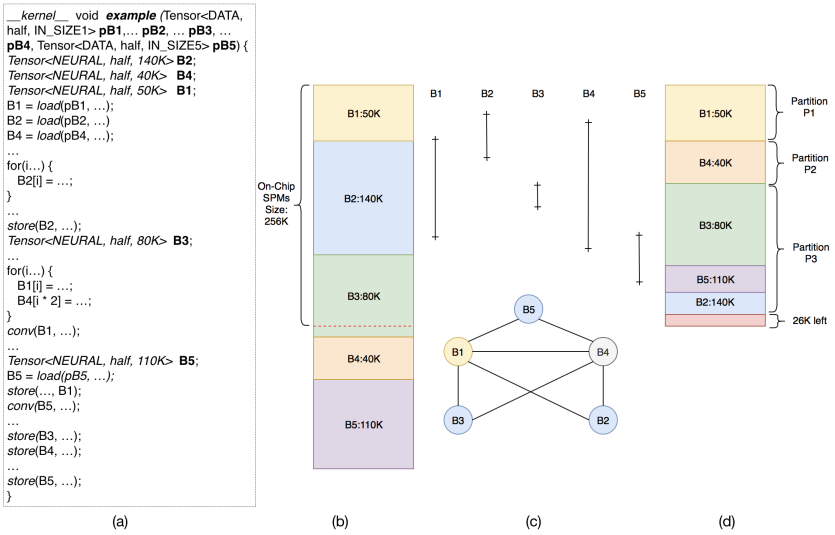


Fig. 4: A motivation example of on-chip scratch pad memory reuse.

block (abbr. BB) $\mathcal{B}$:

$$\begin{aligned} IN_{\mathcal{T}}(\mathcal{B}) &= (OUT_{\mathcal{T}}(\mathcal{B}) - DEF_{\mathcal{T}}(\mathcal{B})) \cup USE_{\mathcal{T}}(\mathcal{B}) \\ OUT_{\mathcal{T}}(\mathcal{B}) &= \cup_{\mathcal{S} \in succ(\mathcal{B})} IN_{\mathcal{T}}(\mathcal{S}) \end{aligned} \tag{1}$$

where $succ(\mathcal{B})$ denotes the set of all successor BBs of $\mathcal{B}$ in CFG of $\mathcal{K}$. The predicates, **DEF** and **USE**, local to a BB $\mathcal{B}$ for a tensor $\mathcal{T}$ are defined as follows: $\mathbf{USE}_{\mathcal{T}}(\mathcal{B})$ returns true if some elements of $\mathcal{T}$ are read in $\mathcal{B}$; $\mathbf{DEF}_{\mathcal{T}}(\mathcal{B})$ returns true if $\mathcal{T}$ is killed in $\mathcal{B}$. Fig. 4c top shows the live ranges of tensor variables $B1$-$B5$. Then compiler builds interference graph (IG). However, considering the memory hierarchies of DNN accelerators, we need to build IG for different address space respectively, i.e., vertices in the same IG must have the same address space specifier. Fig. 4c bottom shows the IG of tensor variables in address space NEURAL. Now compiler takes a greedy graph coloring strategy [4] by clustering the memory blocks with non-overlapping live ranges. Memory blocks with the same color will be assigned to the same memory partition. The size of each memory partition is calculated by choosing the maximum size of colored memory blocks. Fig. 4d shows the memory partitions after graph coloring and there is $26K$ on-chip memory left. Compiler will try to promote some off-chip data (including but not limited to local variable or stack data) into this region. The promotion strategy can be performed by static analysis.

Table 3: Hardware specifications of experimental platform.

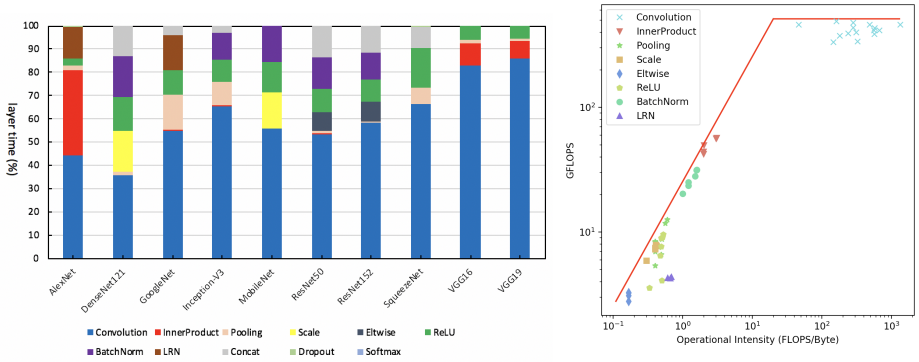
On-chip SPMs	Off-chip DRAM	Peak Performance	Memory Bandwidth
Neural buffer: 512 KB Synapse buffer: 1 MB	8 GB	0.5 TFLOPS	25.6 GB/s

4 Performance Evaluation

4.1 Benchmarks and Baselines

Our experiment is conducted on a prototype accelerator (Table 3). Its architecture refers to the design of Cambricon-ACC [15]. We select 55 typical operators

from popular DNNs as our benchmarks. It covers some representative algorithms used in DNNs, e.g., convolution (conv), fully connection (fc), pooling (pool), scale, element-wise (ew), batch normalization (bn), local respond normalization (lrn) and ReLU activation (relu). All these benchmarks can be expressed with the high-level algorithmic primitives of TOpLib. To show the representativeness of our benchmarks, we have profiled the execution time of some popular DNNs at Cambricon-ACC and decomposed their execution time into typical layers (Fig. 5a). Finally, we extracted 55 most time-consuming operators with typical real-world data scales[3]. We choose the highly tuned hand-written assem-



(a) Breakdown of typical layers' execution time for common DNNs at Cambricon-ACC.

(b) Roofline of the hand-written assembly implementation versions at Cambricon-ACC.

Fig. 5: Layer-wise execution time breakdown of common DNNs and roofline of hand-written assembly implementation of 55 operators at Cambricon-ACC.

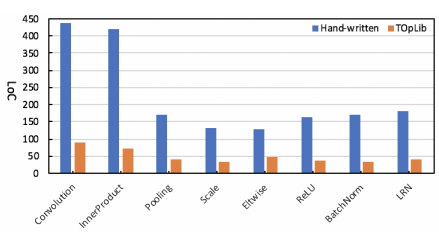
bly implementation of benchmarks as baselines. To show the quality of baselines, we have drawn the roofline of the hand-written assembly implementation versions. From the roofline in Fig. 5b, we can see that these hand-written assembly benchmarks saturate the peak performance of Cambricon-ACC in the terms of FLOPS and memory bandwidth, confirming that *our baselines are very tough to beat.*

4.2 Experimental Results

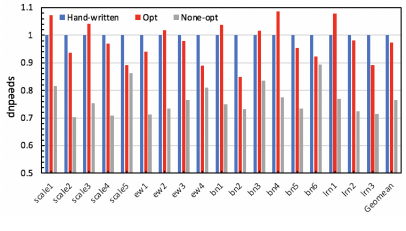
Code Density Code density is a meaningful metric to measure the simplicity and ease of use. We compare code density of TOpLib implementation with the hand-written assembly versions by manually counting the average lines of code (LoC) for each type benchmarks (Fig. 6a). On average, the code length of TOpLib implementation is about 4.9x, 5.8x, 4.3x, 3.8x, 2.7x, 4.4x, 4.8x and 4.6x shorter than the hand-written assembly versions for *conv, fc, pool, scale, ew, relu, bn* and *lrn* operators, respectively.

Execution Performance Fig. 6b, 6c and 6d shows the speedup of TOpLib implementation against hand-written assembly versions. Compared to hand-written assembly versions of all benchmarks, TOpLib achieves 91% on average

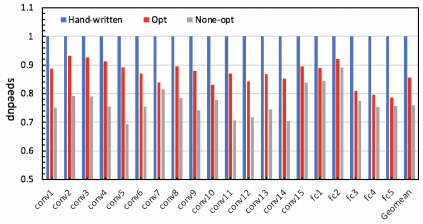
[3] Due to space limit, the detailed data scales are clearly listed in the anonymous github repository: https://github.com/anonymous-0x00/npc20-benchmarks.



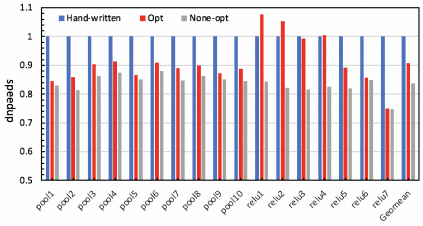
(a) Average code length of TOpLib vs hand-written assembly versions.



(b) TOpLib vs hand-written assembly versions on scale, ew, bn and lrn benchmarks.



(c) TOpLib vs hand-written assembly versions on conv and fc benchmarks.



(d) TOpLib vs hand-written assembly versions on pool and relu benchmarks.

Fig. 6: TOpLib implementation vs hand-written assembly versions on all 55 operators. For (b) (c) (d), the red bar is the normalized execution performance of TOpLib implementation enabling the memory optimization passes, the gray bar disables them.

if we enable the memory optimization passes. However, if we disable them, the execution performance would reduce to about 78% on average. That means memory optimization plays an important role for the producing of high-performance code. Specifically, we find that zip-heavy operators (e.g., scale, ew, bn and lrn) are compiler-friendly for the memory reuse optimization pass.

5 Related Work

This section summarizes the prior work on template library for DNN operators. CUDA [2] or OpenCL [17] is a parallel programming model for GPUs, not for the dedicated DNN accelerators. Eigen [7], Mshadow [6] and Cutlass [18] are linear algebra libraries for CPUs or GPUs. The implementation of existing libraries relies on the meta-programming capability of the C++ programming language, e.g., expression template. They overlook the compiler-assisted optimizations.

6 Conclusion

In this paper, we present TOpLib, a compiler-assisted template library to alleviate low-level programming chores of DNN accelerators. TOpLib provides a user-view abstraction of DNN accelerators, which allows programmers to express operators with high-level algorithmic primitives. We have detailed its design principles and compiler-assisted optimizations. Experimental results show that TOplib could succinctly express the typical DNN operators and produce code that is comparable to hand-written assembly versions.

Acknowledgement. This work is supported by the National Key R&D Program of China (under Grant No. 2017YFB1003103) and the Science Fund for Creative Research Groups of the National Natural Science Foundation of China (under Grant No. 61521092).

References

1. AnandTech: Cambricon, Makers of Huawei's Kirin NPU IP. https://www.anandtech.com/show/12815/cambricon-makers-of-huaweis-kirin-npu-ip-build-a-big-ai-chip-and-pcie-card (2018)
2. Cook, S.: CUDA Programming: A Developer's Guide to Parallel Computing with GPUs. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1st edn. (2012)
3. Cover, T., Hart, P.: Nearest Neighbor Pattern Classification. IEEE Trans. Inf. Theor. **13**(1), 21–27 (Sep 2006)
4. Culberson, J.C.: Iterated Greedy Graph Coloring and the Difficulty Landscape. Tech. rep. (1992)
5. Deng, J., Dong, W., Socher, R., Li, L.J., Li, K., Fei-Fei, L.: ImageNet: A Large-Scale Hierarchical Image Database. In: CVPR09 (2009)
6. DMLC teams: mshadow. https://github.com/dmlc/mshadow (2018)
7. Guennebaud, G., Jacob, B., et al.: Eigen v3. http://eigen.tuxfamily.org (2010)
8. He, K., et al.: Deep residual learning for image recognition. CoRR **abs/1512.03385** (2015)
9. Hearst, M.A.: Support Vector Machines. IEEE Intelligent Systems **13**(4), 18–28 (Jul 1998)
10. Howard, A.G., et al.: MobileNets: Efficient Convolutional Neural Networks for Mobile Vision Applications. CoRR **abs/1704.04861** (2017)
11. Iandola, F.N., et al.: SqueezeNet: AlexNet-level accuracy with 50x fewer parameters and <1MB model size. CoRR **abs/1602.07360** (2016)
12. Jouppi, N.P., et al.: In-Datacenter Performance Analysis of a Tensor Processing Unit. pp. 1–12. ISCA '17, ACM, New York, NY, USA (2017)
13. Krizhevsky, A., et al.: ImageNet Classification with Deep Convolutional Neural Networks. pp. 1097–1105. NIPS'12, Curran Associates Inc., USA (2012)
14. Lian Li, et al.: Memory coloring: a compiler approach for scratchpad memory management. In: PACT'05. pp. 329–338 (Sep 2005)
15. Liu, S., et al.: Cambricon: An Instruction Set Architecture for Neural Networks. pp. 393–405. ISCA '16 (2016)
16. Muchnick, S.S.: Advanced Compiler Design and Implementation. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA (1998)
17. Munshi, A., Gaster, B., Mattson, T.G., Fung, J., Ginsburg, D.: OpenCL Programming Guide. Addison-Wesley Professional, 1st edn. (2011)
18. NVIDIA teams: Cutlass. https://github.com/NVIDIA/cutlass (2017)
19. Progsch, J., et al.: A New Vectorization Technique for Expression Templates in C++. CoRR **abs/1109.1264** (2011)
20. Simonyan, K., Zisserman, A.: Very Deep Convolutional Networks for Large-Scale Image Recognition (2014), https://arxiv.org/abs/1409.1556
21. Springer, M., Sun, Y., Masuhara, H.: Inner Array Inlining for Structure of Arrays Layout. pp. 50–58. PLDI, ARRAY 2018, ACM, New York, NY, USA (2018)
22. Szegedy, C., et al.: Going deeper with convolutions. In: Computer Vision and Pattern Recognition (CVPR) (2015), http://arxiv.org/abs/1409.4842
23. Szegedy, C., et al.: Rethinking the inception architecture for computer vision. CoRR **abs/1512.00567** (2015)
24. Wu, J., et al.: Gpucc: An Open-Source GPGPU Compiler. p. 105–116. CGO'16