swTVM: Exploring the Automated Compilation for Deep Learning on Sunway Architecture

Changxi Liu
School of Computer Science and Engineering, Beihang University
changxi.liu@buaa.edu.cn

Hailong Yang
School of Computer Science and Engineering, Beihang University
hailong.yang@buaa.edu.cn

Rujun Sun
State Key Laboratory of Mathematical Engineering and Advanced Computing
sun.rujun@meac-skl.cn

Zhongzhi Luan
School of Computer Science and Engineering, Beihang University
07680@buaa.edu.cn

Lin Gan
Department of Computer Science and Technology, Tsinghua University
lingan@tsinghua.edu.cn

Guangwen Yang
Department of Computer Science and Technology, Tsinghua University
ygw@tsinghua.edu.cn

Depei Qian
School of Computer Science and Engineering, Beihang University
depeiq@buaa.edu.cn

ABSTRACT

The flourish of deep learning frameworks and hardware platforms has been demanding an efficient compiler that can shield the diversity in both software and hardware in order to provide application portability. Among the exiting deep learning compilers, *TVM* is well known for its efficiency in code generation and optimization across diverse hardware devices. In the meanwhile, the Sunway many-core processor renders itself as a competitive candidate for its attractive computational power in both scientific and deep learning applications. This paper combines the trends in these two directions. Specifically, we propose *swTVM* that extends the original *TVM* to support ahead-of-time compilation for architecture requiring cross-compilation such as Sunway. In addition, we leverage the architecture features during the compilation such as core group for massive parallelism, DMA for high bandwidth memory transfer and local device memory for data locality, in order to generate efficient code for deep learning application on Sunway. The experimental results show the ability of *swTVM* to automatically generate code for various deep neural network models on Sunway. The performance of automatically generated code for AlexNet and VGG-19 by *swTVM* achieves 6.71× and 2.45× speedup on average than hand-optimized OpenACC implementations on convolution and fully connected layers respectively. This work is the first attempt from the compiler perspective to bridge the gap of deep learning and high performance architecture particularly with productivity and efficiency in mind. We would like to open source the implementation so that more people can embrace the power of deep learning compiler and Sunway many-core processor.

KEYWORDS

Sunway architecture, Deep Learning, Automatic Compilation

1 INTRODUCTION

Currently, deep learning has achieved outstanding performance in many fields, including self-driving car [2], face detection [26] and machine translation [7]. The deep learning frameworks such as Caffe [11], TensorFlow [1], PyTorch [12] and MxNet [3], provide an efficient platform to support the research and development on intelligent applications. In the meanwhile, emerging deep learning algorithms exhibit increasing demands for massive computation power. To satisfy the computation demand, various accelerating hardwares such GPU and FPGA have been applied in the deep learning field. Current deep learning frameworks almost rely on the high performance libraries such as cuDNN [6] and MKL [22], which are provided by the hardware vender to accelerate the deep learning application. With new deep learning algorithms and hardwares arising rapidly, the engineering cost for porting the algorithm to the hardware, has increased dramatically. It is necessary to find a way to deploy these emerging deep learning algorithms on the various underlying hardwares automatically and efficiently.

To address the above problem, the end-to-end compiler for deep learning application is proposed. The state-of-the-art deep learning compilers include Glow [19], nGraph [8], Tensor Comprehension [21] and TVM [4]. Take TVM for an example, it adopts the design of two level optimization to automatically generate the code for deep neural network, designed on different deep learning frameworks, to various hardware devices such as CPU, GPU and FPGA. On graph level, TVM applies multiple optimizations to the computation graph derived from the deep neural network, such as operator fusion and data layout transformation. On operator level, TVM converts the computation on operators to the tensor operations targeting the specific hardware architecture, and hides the memory latency by optimizing the instruction pipeline. Moreover, TVM can optimize the code generation automatically according to the shape and data layout of the input to each layer for better performance.

Meanwhile, for its compelling computation power, Sunway many-core processor serves as the basic building block of Sunway TaihuLight supercomputer, which is the first supercomputer to achieve over 100 petaFlops in the world. The Sunway SW26010 processor consists of four core groups (CG). Each CG, including a Management Processing Element (MPE) and 64 Computing Processing Elements (CPEs), can achieve 765 GFlops peak performance in double-precision. The memory attached to each CG is 8GB with bandwidth of 34.1GB/s. The MPE is a complete 64-bit RISC core, typically used for task control and management, whereas the CPE is also a 64-bit RISC core but with limited functionalities, typically used for computation. In addition, each CPE has a 64KB local device memory (LDM), that is managed explicitly by software. The executables on Sunway are generated through cross-compilation with MPE and CPE as different compilation targets. Due to the limitation of Sunway customized operating system, the dynamic linked libraries are not supported.

To embrace the advantage of automatic compilation and high performance for deep learning application, it is intuitive to adopt TVM to Sunway architecture. However, the unique compilation environment and architecture features prevent an naive adoption of TVM to Sunway. First, TVM relies on dynamic link libraries to generate executables on different hardware devices, which is not supported on Sunway. In addition, its code organization fails to recognize the different compilation targets for MPE and CPE, and thus incapable to manage the function calls between MPE and CPE. Second, the memory capacity of each CG on Sunway is quite limited. During the deep learning computation, large memory capacity is required to store the intermediate data as well as the weight parameters. How to allocate the memory space efficiently and leverage the unique architecture features such as DMA for high bandwidth data transfer is important to generate code with high performance. Third, each CPE within a CG contains a 64KB LDM that can be used to buffer data with explicit software management. How to leverage the limited LDM on each CPE with improved data locality is critical for realizing the performance advantage of Sunway during code generation.

To overcome the above challenges, we propose *swTVM*, a deep learning compiler on Sunway that is tailored for the unique compilation environment and architecture features on Sunway. In *swTVM*, we provide ahead-of-time (AOT) code generation that manages the function calls as well as compilation for MPE and CPE explicitly. In addition, we apply several optimizations to the tensor operations so that the architecture features such as DMA and LDM are better utilized during code generation. To the best of our knowledge, this is the first work to implement an end-to-end deep learning compiler on Sunway. Our experimental results show that the automatically generated code for AlexNet and VGG-19 by *swTVM* achieves 6.71× and 2.45× speedup on average for convolution and fully connected layer respectively compared to hand-optimized OpenACC implementations. Specifically, this paper makes the following contributions:

- We implement the ahead-of-time (AOT) code generation, that produces different compilation targets for MPE and CPE as well as manages the function calls between MPE and CPE efficiently. In addition, we manage the intermediate memory space for each tensor operation globally, which avoids the overhead of frequent memory allocation during computation.
- We apply several optimizations to the tensor operations regarding the unique architecture features on Sunway. Specifically, we propose a DMA control interface that manipulates the DMA data transfers for each tensor during computation. In addition, we design a LDM management mechanism that buffers the tensor data as much as possible to reduce the latency for accessing memory. Moreover, the DMA instructions are automatically inserted during code generation to improve the re-accessibility of the buffered data.
- We propose *swTVM* that implements AOT code generation and architecture specific optimizations on top of *TVM*, which offers the high performance of Sunway processor to the deep learning community through automatic compilation. We compare the performance of *swTVM* for several deep neural networks with hand-optimized implementations and the result shows the performance of *swTVM* is even better than hand-optimized OpenACC implementations.

The remainder of this paper is organized as follows. In Section 2, we present the background of the deep learning compiler and Sunway architecture. Section 3 presents the design overview of *swTVM*. Section 4 and Section 5 describe the details of code generation in AOT mode and optimizations for tensor operations on Sunway. Section 6 presents the performance results of AlexNet and VGG-19 compared to hand-optimized implementations using OpenACC and *swCaffe*. Section 7 presents the related work in the fields of deep learning complier and performance optimization on Sunway. Section 8 concludes this paper.

2 BACKGROUND

2.1 Sunway Architecture

The architecture of Sunway processor is shown in Figure 1(b). The Sunway SW26010 processor consists of four core groups (CGs), and each CG includes one Management Processing Element (MPE) and 64 Computing Processing Elements (CPEs). Each CG can achieve 765GFlops peak performance in double-precision and 34.1GB/s theoretical memory bandwidth. The MPE is a complete 64-bit RISC core, designed for task management and control, whereas the CPEs are also 64-bit RISC cores but with limited functionalities, focusing on computation.

The executables on Sunway are generated through cross-compilation on x86 processor using customized compiler (*sw5cc* for C and *sw5CC* for C++). Due to the limitation of the customized operating system on Sunway, it does not support dynamic linked libraries. Instead, the executables are generated with libraries statically linked. Moreover, the code running on MPE and CPEs is complied as different compilation targets on Sunway. For instance, with Sunway C compiler, additional compilation options are added to generate the executable for MPE (*-host*) and CPEs (*-slave*) respectively.

The memory hierarchy on Sunway processor is also different from x86 processor. Each MPE has a 32KB L1 data cache and a 256KB L2 data/instruction cache, whereas each CPE has a 16KB L1 instruction cache and a 64KB local device memory (LDM). The LDM is commonly used as a programmable buffer with explicit software control. There are two ways to access memory on Sunway. The first one is to use DMA, which prefers large and continuous data access. The other one is to use global load/store (Gload/Gstore) instruction, which prefers small and random data access.

Two parallel programming models are supported on Sunway to exploit the massive parallelism of the CPEs, including OpenACC and Athread. The OpenACC programming model is more programmer friendly. With OpenACC, programmers can utilize different processors without knowing about the underlying architecture details. Whereas, Athread is the parallel programming model tailored for Sunway architecture. With Athread, programmers can buffer the data in LDM, which provides the opportunity to reduce the accesses to main memory through explicit control. In this paper, we use Athread to generate the code on Sunway for better performance.

Although the LDM of CPE sounds similar to the shared memory on GPU, the design philosophy is quite different between them. GPU adopts SIMT parallelism that accesses the shared memory through concurrent threads within a warp. The GPU program achieves better performance if threads within a warp access a continuous memory region at the same time. However, on Sunway the CPEs access the memory and buffer the data in LDM independently. Therefore, without careful management, severe contention on memory bandwidth would occur and thus degrade the performance significantly. In addition, when buffering large continuous data block to LDM, the DMA data transfer can be utilized for higher memory bandwidth.

2.2 Automated Compilation for Deep Learning

Recently, the deep neural network has obtained significant achievement on self-driving, face detection and machine translation. With the increasing accuracy and application domain, the tendencies for the future development of deep neural network become clear: *1)* the size of the neural network scales with deeper layers; *2)* the algorithms of the neural network become more diverse; *3)* new hardwares are emerging to accelerate the computation of neural network; *4)* various deep learning frameworks are adopted by the developers.

The above tendencies increase the demand for deploying the emerging neural network algorithms to various hardware devices, so that enormous engineering efforts are required to match the algorithms with the hardwares efficiently. Currently, the performance of the neural network mainly depends on the computation library such as cuDNN and MKL provided by hardware vendors. However, it is unsustainable to perform labor intensive performance tuning to match various hardwares as new algorithms are arising rapidly. The deep learning compiler provides a way to build an efficient mapping between new algorithms and various hardware devices, and thus improves the portability of neural network algorithms. Pioneers attempt to build deep learning compiler for the entire community including TVM, Glow and nGraph.

Despite different implementation approaches adopted by different deep learning compilers, their design philosophies (e.g., two level optimization) are somehow converging. Therefore, we take TVM for an example to illustrate. TVM uses the idea of two level optimization including graph level and operator level. On graph level, it converts the deep neural network to the computation graph, and then applies optimizations such as operator fusion and data layout transformation. On operator level, it optimizes the code generation targeting specific hardware through loop fusion and loop re-ordering. However, adopting existing deep learning compiler such as TVM, to Sunway introduces several challenges to be addressed, regarding the unique compilation environment and architecture features on Sunway. These challenges motivate our work in this paper.

2.3 Challenges for DL compilation on Sunway

The first challenge is that Sunway processor relies on cross-compilation to generate executables and does not support dynamic linked libraries. These limitations prohibit naive adoption of existing deep learning compiler such as TVM to Sunway. Therefore, code generation in AOT mode needs to be supported in the deep learning compiler so that it can compile the executables with static linked libraries. In addition, an efficient code organization is required with AOT code generation in order to support different compilation targets for MPE and CPE as well as function calls between MPE and CPE. For instance, the management and control code should be generated on MPE, whereas the computation code should be generated on CPEs. Moreover, the memory capacity of a CG is quite limited compared to the large volume of data generated during the tensor operation. To avoid the overhead of frequent memory allocation during computation, the memory needs to be managed globally in AOT code generation.

The second challenge is to optimize the generated code regarding the unique architecture features of Sunway. Observed by existing research works [15–17], the key to achieve high performance on Sunway is to *1)* fully utilize the computing resources of CPEs for massive parallelism, and *2)* leverage the LDM of each CPE to alleviate the bottleneck of memory access. Therefore, when the neural network compiler optimizes the generated code, the following three rules need to be complied: *1)* use the DMA as much as possible when accessing main memory. The DMA requires accessing large and continuous data block, which provides higher memory bandwidth; *2)* leverage the LDM to buffer as much data as possible during the computation. The LDM reduces the latency to access main memory; *3)* minimize the frequency of memory access as much as possible. The computation should exhibit better data locality and re-accessibility after each memory access.

In sum, implementing an end-to-end compiler for deep learning application requires not only adaptions to the compilation environment on Sunway, but also optimizations targeting the architecture features to improve the performance of generated code.

3 *SWTVM*: DESIGN OVERVIEW

To address the challenges described in Section 2.3, we implement the AOT code generation as an extension to *TVM*, and manage the code organization for MPE and CPE respectively. In addition, we manipulate the memory allocation of the tensor operation globally. The grey components in Figure 1(a) show the contribution of our work. To improve the readability and portability of the generated code, we produce C source code in AOT mode, which is then compiled by Sunway native compiler in order to generate the executables. The advantage of AOT code generation is that, the memory allocation for each layer is determined based on the input and output of each layer before the actual computation, which avoids frequent

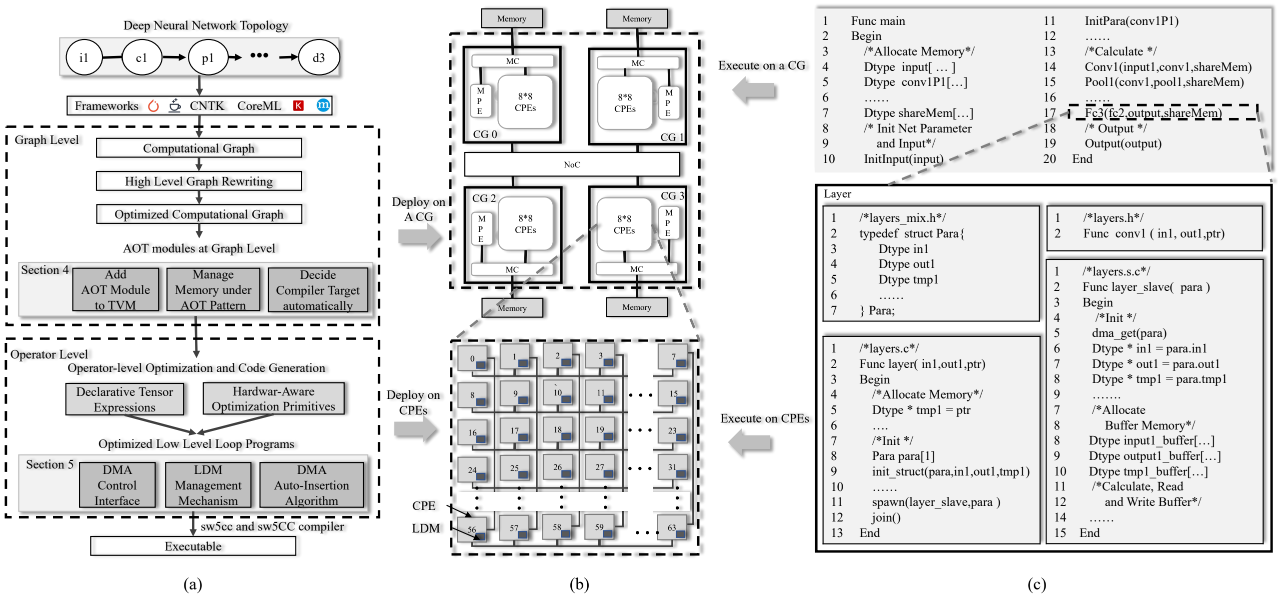

Figure 1: (a) The design overview of *swTVM*, (b) the Sunway architecture and (c) an illustrative example of automatically generated code for deep neural network.

memory allocation during the computation and thus eliminates the overhead of operations related to memory allocation.

Since the MPE supports complete functionality, the generated code in AOT mode can run on MPE directly. Whereas for CPE, we need to generate a separate file for compiling on CPE, as shown in Figure 1(c). We use structure to accept multiple parameters in the function on CPE. In order to remove the dependency on the structure definition from the interface when calling the layer, we encapsulate CPE functions with another interface that renders the layer function calls as ordinary function calls. The encapsulating interface is also useful when handle the memory allocation of the intermediate data for complex layers. The encapsulated function is organized in a separate file with MPE as its compilation target. The parameters stored in the structure file is only visible to the files containing the CPE function and encapsulated CPE function. We achieve the AOT code generation for each layer by organizing the code of each layer into the above three files in addition to a header file for the encapsulated CPE function.

The MPE code generated in AOT mode on is primarily responsible for calling each layer according to the topology of the neural network, whereas the code generated for CPE is responsible for the specific computation of each operation. We optimize the implementation of each operator adopting to the architecture features on Sunway, as shown in Figure 1(a). Specifically, we design a DMA control interface, which can provide the DMA schedule primitive for high-level language such as Python. In addition, the LDM on each CPE is only 64KB and the amount of tensor data is too large to fit in entirely. To decide the size of the tensor data accessed through DMA and buffered in the LDM, we propose a LDM management mechanism to control the number and the size of the tensor data to be buffered automatically, which can also adjusts to the different configuration of input and output of the neural network. Moreover, to improve the re-accessibility of the buffered data, the DMA instructions need to be inserted to the right place of the generated code. To achieve this, we design an algorithm to insert DMA instructions into the optimal locations with improved data re-accessibility of the generated code automatically.

4 AOT CODE GENERATION

To implement AOT code generation, we should consider not only the implementation of each layer, but also how to convert the neural network topology into the function calls of each layer with the dependence among each layer satisfied. The Figure 1(a) shows the topology of a deep neural network consists of convolution and fully connected layers. After AOT code generation, the implementation is shown in Figure 1(c) as a serial of operations such as memory allocation, parameter initialization and function calls in *Func main*.

4.1 Managing Memory Allocation

When generate code in AOT mode, the memory allocation on both main memory and LDM for input/output data as well as temporal variables of each layer needs to be managed. The memory allocated for input/output data includes intermediate data generated between each layer as well as weight parameters, which cannot be released or override during computation. After completing computation, each operation stores its result into memory which is then used by other operations. As these data is stored in main memory, the memory space is allocated and released by MPE. The implementation details are shown in Figure 1(c).

For complex operations, the computation also produces temporal data. These data is never used again when the operation completes. Therefore, the memory space for temporal data can be released or override. However, the size of the temporal data is easily larger than the capacity of LDM, it is also stored to main memory. The memory space for the temporal data is allocated and released in the main function. When an operation is invoked, it uses part of the memory space that has already been allocated in the main function, which reduces the overhead for allocating and releasing memory space across each operation. The memory space for temporal data is allocated by MPE and used by CPE. The implementation details are listed in *Func main* for MPE and *Func Layer_slave* for CPEs in Figure 1(c). The utilization of LDM in *Func Layer_slave* is described in Section 5.

4.2 Managing Function Call

As shown in Figure 1(a), the implementation of *swTVM* compiler is organized into three levels, which first transforms the topology of the deep neural network into computation graph, and then applies a serial of optimizations at graph level, and eventually implements the computation on specific hardware at operation level. In AOT code generation, the *Func main* in Figure 1(c) is responsible for maintaining the dependency of function calls in the computation graph, whereas *Fun layer_slave* in Figure 1(c) implements each operation. Note that the *Func layer* in Figure 1(c) connects *Fun layer_slave* and *Func main*, and fulfills the function call of each operation in the computational graph.

In addition, function calls for the architecture specific code can also be organized into three levels, including function call on MPE, function call on CPE and function call from MPE to CPE. These three levels map back to the operation call at the graph level, operator level and from graph level to operator level in the implementation of *swTVM*. The graph level generates *Func Main* shown in Figure 1(c), which runs on MPE. And the *Func layer* shown in Figure 1(c) is the implementation of the function call from graph level to operator level, which is called by *Func main* on MPE and then calls the *Func layer_slave* to run on CPE. *Func layer_slave* implements the computation invoked at the operator level.

The advantage of such design is that we can implement *swTVM* through the integration of AOT code generation and Sunway architecture optimizations through function call, without relying on the implementation details of each level. Using AOT compilation, *swTVM* does not need consider the underlying implementation when generating the operations. In addition, through managing the dependency of function calls, *swTVM* is able to generate code for MPE and CPE as different compilation targets.

4.3 Implementation Details

The *Func main* shown in Figure1(c) consists of four stages, including memory allocation stage, parameter/input initialization stage, computation stage and output stage. During memory allocation stage, in addition to allocate memory for the parameter and input/output of the neural network, a temporal memory space is also allocated for each layer, the size of which satisfies the maximum memory usage across each layer. The dependency across all network layers is analyzed to ensure the order of function calls is in accordance with the dependency. Each function in the computation stage corresponds to one or more layers in the neural network topology.

The implementation of each operation consists of *Func layer* on MPE, *Func layer_slave* on CPEs and parameter structure *Para*. *Func layer* is further divided into three parts such as the memory allocation for temporal space, parameter initialization and computation. The memory allocation for temporal space is only required for layer that combines multiple sub-operations such as convolution and pool layer. For such layer, the input of one sub-operation depends on the temporal results from the previous sub-operation. Due to performance overhead for frequent memory allocation, we allocate a temporal memory space in the main function and share it across each layer.

The format for calling function on CPE is to use the function name and parameter structure as shown in Figure1(c). *Para* is the parameter structure that is only visible to corresponding *layer.c* and *layer.slave.c* files. *Func layer_slave* consists of parameters parsing, LDM allocation and computation. At the beginning of the function, the tensor data is loaded from memory and then buffered in LDM. The memory of LDM is allocated through static arrays to buffer tensor data. The memory is accessed through DMA instructions and overlapped with the computation to reduce the latency, the details of which are described in Section 5.

5 OPTIMIZING TENSOR OPERATION

5.1 DMA Control Interface

An efficient DMA control interface plays an important role for *swTVM* to generate high performance implementation of neural network on Sunway. The DMA control interface provides schedule primitives for the high level language such as Python to control the DMA instruction in order to manage the data access efficiently. The Figure 2 shows how to control the tensor data access in matrix multiplication through the DMA control interface. As seen, we are able to specify not only which tensor to be buffered, but also the region of the tensor to be buffered during code generation. When partial of the data along one dimension of the tensor needs to be buffered, *split*, *buffer_read* and *buffer_write* primitives are applied in turn to split the dimension and buffer the corresponding data. After invoking the above primitives, *Load Data* region generates the IR code of read buffer for tensor B and A, shown in Figure 2(d). Whereas *Store Data* region generates the IR code of write buffer for tensor C. These IRs are then translated to DMA instructions during code generation.

swTVM can buffer data not only along one dimension, but also along multiple dimensions. In Figure 2(c), the tensor B is buffered along two dimensions. Because the value along these two dimensions of tensor B is required when calculating the region of tensor C to be buffered. Buffering along multiple dimensions also occurs in convolution. The convolution is computation among high dimensional tensors, however certain dimensions of the tensor are quite small, for instance the rx and ry of tensor W in Figure 3. If buffering data along only one dimension, the memory is not fully utilized. In such case, buffering the tensor data along multiple dimensions improve the bandwidth utilization. When buffering, we satisfy the data need of outer loop with priority, which improves the re-accessibility of the buffered data.

In complex layer such as convolution, the subscript to access the tensor data along one dimension is determined by multiple variables. To handle such case, the DMA control interface accepts multiple variables and allows the user to specify how to construct

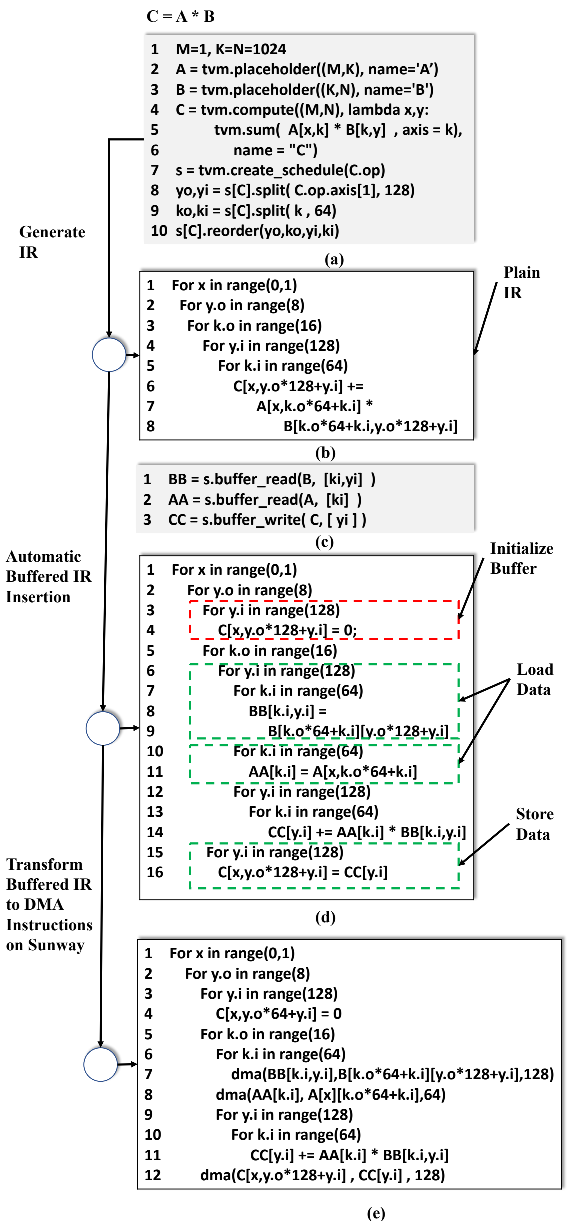

Figure 2: An example of generating matrix multiplication implementation that is optimized on Sunway.

B = Conv(A , W)

```
1  stride = 2
2  ......
3  B  = tvm.compute( ( channel, height, width),
4         lambda ff,yy,xx:  tvm.sum(
5             A( rc , yy * stride + ry , xx * stride + rx ) * W[rc , ry , rx] ) ,
6                 axis = [rc , ry , rx ] , name = "B")
7  s = tvm.create_schedule(B.op)
8  AA = s.buffer_read(A,  [yy*stride+ry,xx*stride+rx])
9  ......
```

Figure 3: An example of generated convolution implementation on Sunway.

Algorithm 1 LDM management algorithm.

```
function LDMManagement(itervars, tensorset)
    /*Classify itervars to sizevars, numvars and compvars*/
    {sizevars, numvars, compvars} ←
        CLASSIFY(itervars, tensorset)
    for var ∈ itervar do
        Buffer(var) ← 1
    Sort(compvars)
    Sort(sizevars)
    Vars = {sizevars, compvars}
    /* initial buffer size */
    while Vars ≠ {} do
        sizevars ← {}
        compvars ← {}
        for iter ← 0, LEN(vars) do
            var ← vars(iter)
            if range(var) < 64 then
                Buffer(var) ← range(var)
                UPDATE(itervars, tensorSet, var, UP)
            else
                Buffer(var) ← 64
                while dma_use > dma_size do
                    if Buffer(var) == range(var) then
                        UPDATE(itervars, tensorSet, var, DOWN)
                    Buffer(var) ← Buffer(var)/2
                    if Buffer(var) == 0 then
                        Buffer(var) ← 1
                        iter ← iter − 2
                        Break
        sizevars, numvars, compvars ←
            CLASSIFY(itervars, tensorset)
        Sort(compvars)
        Sort(sizevars)
        Vars = {sizevars, compvars}
    /* expand buffer size */
    while True do
        var = select(Vars)
        Buffer(var) ← Buffer(var) * 2
        if dma_use > dma_size then
            Buffer(var) ← Buffer(var)/2
            Break
        if Buffer(var) == range(var) then
            UPDATE(itervars, tensorSet, var, UP)
            {sizevars, numvars, compvars} ←
                CLASSIFY(itervars, tensorset)
            Sort(compvars)
            Sort(sizevars)
            Vars = {sizevars, compvars}
```

the subscript using these variables in order to determine the range of each dimension to be buffered. One such example is shown in the expression $yy * stride + ry$ (line 8) of Figure 3. The DMA control interface also supports expression parsing, which accepts the subscript expression and analyzes the correlation between the variables and tensor dimensions automatically.

5.2 LDM Management Mechanism

To better control the data buffering in LDM, we design a LDM management mechanism, which determines the dimensions of tensor to be buffered and the buffer size in LDM. In addition, it re-orders the computation loop in order to improve the re-accessibility of the buffered data.

The Algorithm 1 shows the algorithm for managing data buffer in LDM. It uses an approximate algorithm to ensure the search time for an optimal solution is not too long. The algorithm consists of two parts, the initial part to allocate a pre-defined LDM memory space

Algorithm 2 Loop variable re-ordering algorithm.

```
1: /*select the var which requires the least number of DMA transfers*/
2: function SELECT(Vars)
3:     cur_var ← NULL
4:     cur_dmatimes ← INTMAX
5:     for varid ← 0, LEN(Vars) do
6:         var ← Vars(varid)
7:         dmatimes ← count(var)
8:         if cur_dmatimes < dmatimes then
9:             cur_var ← var
10:            cur_dmatimes ← dmatimes
11:    return cur_var
12: function ReorderLoop(buffervars, vars)
13:    varorder ← []
14:    /* Classify itervars to buffervars and vars */
15:    varorder.add(buffervars)
16:    while true do
17:        var ← SELECT(vars)
18:        if Var ≠ NULL then
19:            varorder.add(var)
20:            vars.rm(var)
21:        else
22:            break
23:    return varorder
```

Algorithm 3 DMA auto-insertion algorithm.

```
function AutoInsertDMA(itersets, TensorSet, BufTensorSet)
    /* Analyze correlation between tensor and variable */
    itersSet ← AnalyzeCorrelation(itersets, TensorSet)
    SubItersSet ← AnalyzeCorrelation(itersets, BufTensorSet)
    /* Find the location to insert DMA instruction*/
    for iter ∈ itersets do
        /*insert DMA instruction*/
        for tensor ∈ TensorSet do
            if ItersSets(tensor) − SubItersSet(tensor) == {} then
                InsertDMA(tensor)
                TensorSet.rm(tensor)
        /*update the itersSet for each tensor*/
        for tensor ∈ TensorSet do
            TensorIters ← ItersSet(tensor)
            SubTensorIters ← SubItersSet(tensor)
            if iter ∈ SubTensorIters&&iter ∉ SubTensorIters then
                TensorIters.rm(iter)
            else
                if iter ∉ TensorIters then
                    Continue
                else
                    Raise(Error)
```

and expanding part to maximize the LDM utilization. In Algorithm 1, the variables can be classified into three types:

- *sizevar*: determines the buffer size and is the index of the lowest dimension of the tensor that can expand buffer size;
- *numvar*: determines the number of DMA instructions and is the index of the dimension (except the lowest dimension which can expand buffer size) of the tensor;
- *compvar*: satisfies the conditions of both *sizevar* and *numvar*.

At the beginning of the algorithm, the sequence of the variables is re-ordered. For *compvars*, it is re-ordered by the ascending order of the affected number of tensors. Whereas for *sizevars*, it is re-ordered by the ascending order of the buffer size (line [7-8]). After that, the buffer for each variable is initialized to a pre-defined size across each tensor (line [10-33]). The buffer size is set to the minimum of the variable size and 64. The number 64 is chosen based on empirical study that reading 64 floats per memory access achieves good bandwidth on Sunway (line [16-20]). Then, the algorithm checks if the buffer size is larger than the capacity of LDM. If so, the data to be transferred for current variable or even the previous buffered variable needs to be reduced to fit in the limited capacity of LDM (line [21-28]).

During the initialization, the algorithm invokes the *UPDATE* function if the range of the variable to be buffered equals to the range of the loop variable. When the dimension of the tensor to be buffered is no longer associated with any variables, the higher dimension needs be adjusted to change the buffer size. And the *CLASSIFY* function is invoked to update *numvars*, *sizevars* and *compvars* (line [22-23, 29-33, 41-47]). After the initialization, if the LDM still has free space, the buffer size of each variable is expanded to improve the LDM utilization. We use a greedy algorithm to load as much data into LDM as possible. The algorithm terminates when the LDM usage reaches the maximum capacity (line [35-47]).

We take the matrix multiplication in Figure 2 as an example to illustrate the process of the algorithm, where x, y and k is *numvar*, *sizevar*, and *compvar* respectively. We set the buffer size of y to 64 and check our buffer size not exceeding the LDM capacity. Then, we set k to 64 that leads to the LDM usage of 16.5KB. Since there is no *numvar* satisfying the condition of *UPDATE*, the algorithm enters the expanding part. When y is set to 128, the LDM usage increases to 32.75KB. Continuing to expand with k set to 128, the buffer size reaches 65KB, which is larger than the LDM capacity. Therefore, $x = 1$, $y = 128$, and $k = 64$ is chosen for the buffer size. The time complexity of Algorithm 1 is polynomial time.

After initializing the buffer size for each tensor, the loop order is adjusted to improve the re-accessibility of the buffered data. To avoid unnecessary DMA transfers, the loop variables that are not associated with the tensor are put in the loop where the DMA instructions reside. For the conflicting DMA instructions, the loops are re-ordered and the one with the least number of DMA instructions is chosen, as shown in Algorithm 2. First, all buffered variables are moved to the innermost loop. And then, the order of non-buffered variables are determined. The buffered variables with locations undecided are inserted to the current loop with the number of DMA instructions for all tensors evaluated. The variable with the least number of DMA instructions is chosen as the loop variable for current loop (line [2-11]). The above process is repeated until there is no variable unevaluated, which derives the final loop order (line [16-22]). The time complexity of Algorithm 2 is also polynomial time.

We take the matrix multiplication in Figure 2 for an example to illustrate the loop re-ordering. The variables for which the order to be decided is *x*, *yo* and *ko*. The least number of DMA instructions for *x*, *yo* and *ko* is 256 + 128, 256 + 16 and 256 + 8 respectively. Therefore, *ko* is chosen first. And then, the least number of DMA instructions for x and yo are both 8. Therefore the original loop order is unchanged. After that, the final loop order for *x*, *yo* and *ko* is determined.

5.3 DMA Auto-Insertion Algorithm

With the DMA control interface and LDM management mechanism available, we propose an algorithm to implement the auto-insertion of DMA instructions during the code generation. The DMA auto-insertion algorithm consists of three parts, *1)* determining the buffer size and the starting memory location of each dimension of the tensor; *2)* determining the DMA insertion location to improve the data re-accessibility; *3)* generating code based on the above information and the Sunway instruction syntax.

Determining the buffer size and the starting memory location. First, the buffer dimension is split into two parts, which makes the range of the inner loop within the buffer size. When the subscript of the dimension is correlated to only one loop variable, the starting memory location of the buffer is calculated by setting the loop variable of the inner loop to 0, whereas the buffer size is the range of the inner loop. All buffer operations in Figure 2(c) belong to the above case. However, for complex operation such as convolution in Figure 3, the subscript of a dimension of the tensor is always correlated to several variables. To obtain the starting memory location of the buffer, all variables are set to zero and calculated in the subscript expression. The size of the buffer is the difference between the result of the subscript expression with all variables set to their maximum value and the starting memory location.

Determining the DMA insertion location. In Algorithm 3, all tensors to be buffered constitute the *TensorSet*. First, the correlation between loop variables and memory accesses for each tensor is analyzed. For each tensor, the correlated loop variables are stored in *itersSet(tensor)*. Similarly, for each buffered tensor, the correlated loop variables are stored in *SubIterSet(tensor)*. *SubIterSet(tensor)* is a subset of *IterSet(tensor)* (line [3-4]). After the initialization, the location for each DMA instruction is to be determined. The algorithm checks all locations by iterating through all variables (line [5-22]). When traversing the variable *var*, it checks if $ItersSet(tensor) - SubItersSet(tensor)$ is an empty set. If so, it means that all variables the tensor depends on are determined, and the DMA instructions for the tensor should be inserted in current loop. After that, the tensor is removed from the *TensorSet* (line [8-11]), and the *IterSets(tensor)* of each tensor is updated. If *var* of current loop is not correlated to the tensor, the algorithm proceeds to next tensor. If *var* belongs to *ItesrSet(tensor)* but does not belong to *SubItersSet(tensor)*, then *var* is removed from *ItesrSet(tensor)* indicating *var* is determined (line [12-22]). When Algorithm 3 terminates, the locations to insert DMA instructions for each tensor are determined. The time complexity of Algorithm 3 is polynomial time.

Generating code with Sunway DMA syntax. Figure 2(e) shows the pseudo-code of the inserted DMA instructions. When generating the code, the DMA instructions whose memory address and LDM buffer address is continuous are combined to reduce the number of DMA instructions.

5.4 Implementation Details

Achieving parallelism with CPEs. To achieve good parallelism wit CPEs, the load balance and write conflict need to be considered when generating code on Sunway. The load balance can be achieved using the *athread_parallel* primitive, which splits the high dimension of the tensor across multiple tasks. The Figure 4 shows an example of vector multiplication, $v = v1 \times v2$. For the vector v whose size of dimension is 1024, we divide its dimensions into 64 chunks and the size of each chunk is 16. *_begin* and *_end* indicates the range of tasks for each CPE, which is determined by the *id* of

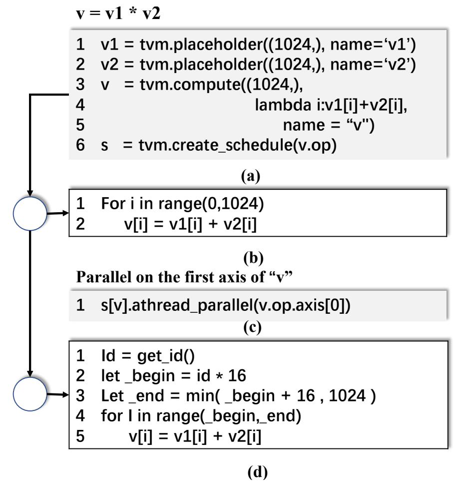

Figure 4: An example of generated parallel implementation on Sunway.

Algorithm 4 Analyze the correlation between tensor and loop variable.

```
1: function AnalyzeCorrelation(LoopVarSets, TensorSet)
2:     /* Initialize the set of variables for each tensor */
3:     for tensor ∈ TensorSet do
4:         VarsSet(tensor) ← {}
5:     /* Analyzing*/
6:     for tensor ∈ TensorSet do
7:         for index ∈ tensor.dims do
8:             for var ∈ index do
9:                 if var ∈ LoopVarSets then
10:                    VarsSet(tensor).add(var)
11:    return VarsSet
```

CPE and the number of the tasks. The worst case happens when the size along the high dimension of the tensor is less than the number of CPEs. Such a case can be solved by using the *fuse* primitive to combine multiple dimensions until the size is large enough. Writing conflict can be avoided by splitting the tasks along the dimension of the tensor to be written.

Analyzing the correlation between tensor and variable. Algorithm 4 illustrates how to derive the correlation between tensors and variables used in Section 5.1 and 5.3. First, there is a set of variables and tensors, which contains the subscript information of each dimension (line [1]). Then, the set of correlated variables for all tensors *VarsSet* is initialized to empty and each tensor in *TensorSet* is analyzed (line [2-10]). When iterating through the *TensorSet*, all variables in the index expression of each dimension is checked and the ones that belong to the set of loop variables are added to *VarsSet* (line [7-10]). When Algorithm 4 terminates, the correlation between variables and tensors is determined.

Loop variable classification. Classified loop variables are used in Section 5.2. The Algorithm 5 describes how the loop variables are classified. First, the loop variables in *varSet* are iterated to obtain their types, which includes *sizeType*, *compType* and *numType*. The type of each loop variable is stored in the corresponding set (line [25-33]). The function *VARTYPE* is used to dervie the type of each loop variable (line [2-20]). When iterating through *tensorSet*, if the

Algorithm 5 Classify the loop variables to *compvars*, *sizevars* and *numvars*.

```
1:  /* classify the type of var */
2:  function VARTYPE(var, tensorSet)
3:      sizeTypeFlag ← false
4:      numTypeFlag ← false
5:      for tensor ∈ tensorSet do
6:          if var ∈ tensor.index(tensor.curBufDim) then
7:              sizeTypeFlag ← true
8:          for dimId ← tensor.curBufDim + 1, tensor.dimSize do
9:              if var ∈ tensor.index(dimId) then
10:                 numTypeFlag ← true
11:     if sizetypeFlag then
12:         if numtypeFlag then
13:             return compType
14:         else
15:             return sizeType
16:     else
17:         if numtypeFlag then
18:             return numType
19:         else
20:             RaiseError
21: function CLASSIFY(varSet, tensorSet)
22:     compvars ← {}
23:     sizevars ← {}
24:     numvars ← {}
25:     for var ∈ varSet do
26:         Type ← VARTYPE(var, tensorSet)
27:         if Type == sizeType then
28:             sizevars.add(var)
29:         else
30:             if Type == numType then
31:                 numvars.add(var)
32:             else
33:                 comvars.add(var)
34:     return {compvar, sizevars, numvars}
35: function UPDATE(varSet, tensorSet, var, direction)
36:     if direction == UP then varSet.rm(var)
37:     elsevarSet.add(var)
38:     for tensor ∈ tensorSet do
39:         if direction == UP then
40:             checkDim ← tensor.curBufDim
41:             while True do
42:                 update ← true
43:                 for var ∈ tensor.index(checkDim) do
44:                     if var ∈ varSet then
45:                         update ← false
46:                 if update then
47:                     checkDim ← checkDim + 1
48:             tensor.curBufDim ← checkDim
49:         else
50:             checkDim ← tensor.curBufDim − 1
51:             for dimId ← 0, checkDim do
52:                 if var ∈ tensor.index(dimId) then
53:                     tensor.curBufDim ← dimId
54:                     break
```

var belongs to the range of current buffered dimension, the value of *sizeTypeFlag* is set to true. And if *var* belongs to the range of higher dimension of the tensor iterated, *numTypeFlag* is set to true (line [3-10]). The return value of function *VARTYPE* is *compType* if the value of *sizeTypeFlag* and *numTypeFlag* are both true. Otherwise, the return value is the type whose flag is true (line [11-20]). The variables in *varSet* are classified by *UPDATE* function in Algorithm 5. The current buffered dimension used in function *CLASSIFY* is updated by function *UPDATE*. First, the *UPDATE* adjusts the set of variables *varSet* whose buffer size is not within the loop range (line [36-37]). If the direction of changing is *UP* and the variable in the current dimension index of the tensor does not exist in *varSet*, the dimension of the current buffered tensor is increased. In contract, the dimension of the current buffered tensor is decreased (line [38-54]).

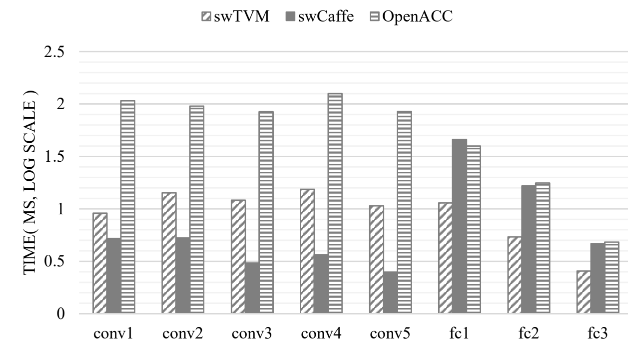

Figure 5: The execution time of each layer in AlexNet implemented using *swTVM*, *swCaffe* and OpenACC respectively.

6 EVALUATION

In this section, we evaluate the performance of the generated code by *swTVM*. We compare our work with hand-optimized implementations using OpenACC and *swCaffe* [14] on AlexNet [13] and VGG-19 [20].

6.1 Experimental Setup

The *swTVM* executes on the x86/64 processor with operating system of CentOS 7.3. The compilation environment includes gcc/g++ 4.8.5 and Python 3.6.2. We set batch size to 1 during the evaluation. The generated code by *swTVM* is a group of C files, which are then compiled by Sunway native compiler (sw5cc for C and sw5CC for C++) with *-O3*.

6.2 AlexNet and VGG Evaluation

We compare the execution time of each layer in AlexNet implemented using *swTVM*, *swCaffe* and OpenACC respectively. The execution time of the convolution and fully connected layer includes the ReLU layer. For the ease of visualization, the value in Y axis is log scaled. As shown in Figure 5, the performance of convolution layer generated by *swTVM* is higher than OpenACC, but lower than *swCaffe*. This is because *swCaffe* applies advanced optimization techniques on Sunway such as register communication and instruction re-ordering, which is not yet supported by *swTVM*. Comparing to OpenACC, *swTVM* achieves better performance due to the efficient DMA data transfer and the utilization of LDM for improved data locality. For convolution layer and fully connected layer, *swTVM* achieves 8.32× and 2.87× speedup on average over OpenACC with the maximum speedup of 11.8× on *conv1* layer.

From Figure 5 we can see that, for fully connected layers, the performance of *swTVM* is better than *swCaffe* and OpenACC. This is because matrix multiplication is much simpler than convolution. The data sharing through register communication adopted by *swCaffe* causes additional overhead and outweighs its benefits. Whereas, *swTVM* can generate code with optimization targeting the specific input/output configuration and thus achieves better

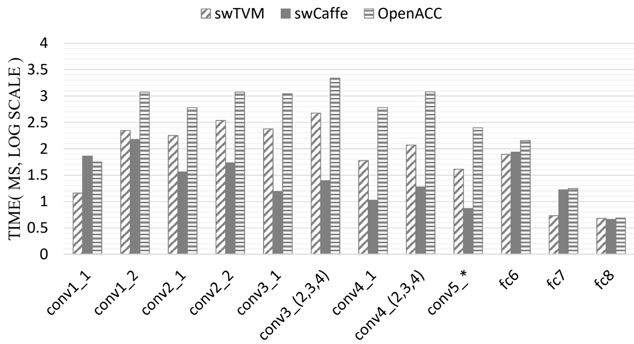

Figure 6: The execution time of each layer in VGG-19 implemented using *swTVM, swCaffe* and OpenACC respectively.

performance. We also notice that the performance of *swCaffe* and OpenACC is similar, which is due to the reason that OpenACC is able to manipulate the DMA data transfer for simple layer such as fully connected layer.

In addition to AlexNet, we evaluate the performance of VGG-19 implemented using *swTVM*, *swCaffe* and OpenACC respectively. Since the design and configuration of *conv3_2*, *conv3_3* and *conv3_4* is the same in VGG-19, we combine them into one layer and average the execution time across original layers. The four convolution layers in *conv5* is processed in the same way. As shown in Figure 6, the performance comparison of the convolution layers and the fully connected layers in VGG-19 is similar to AlexNet across these three implementations.

For the convolution layer, the performance of *swTVM* and *swCaffe* is similar for the two layers on *conv1*. Actually *swTVM* is even better than *swCaffe* on *conv1_1*. This is because the performance of *swCaffe* declines when the size of input and out data is too large to be buffered in LDM. We also notice that even OpenACC is faster than *swCaffe* on *conv1_1*. This is because the number of input channel for *conv1_1* is only three, and the convolution kernels are small enough to be fitted in LDM. The OpenACC implementation takes advantage of the above facts to optimize its performance. However, the performance of *swTVM* is far better than OpenACC due to its optimizations using DMA data transfer, LDM buffer and loop reordering. For convolution layer and fully connected layer, *swTVM* achieves 6.21× and 2.03× speedup on average over OpenACC with the maximum speedup of 10.15× on *conv4_(2,3,4)* layer.

In the meantime, we can notice that the performance of *swCaffe* and *swTVM* is similar in *fc6*. It indicates that *swCaffe* achieves better performance by sharing data through register communication in large matrix multiplication. This inspires us to add register communication support into *swTVM* in order to achieve comparable performance as highly optimized libraries on Sunway.

7 RELATED WORK

7.1 Deep Learning Compiler

Currently, the deep learning community develops rapidly. There are always emerging neural network algorithms and hardware devices. However, the engineering efforts of porting various neural network algorithms to numerous hardware devices increase dramatically. Under this background, the end-to-end neural network compiler is proposed.

The TensorFlow XLA [1] by Google focuses on the high-level computation graph, and it can fuse those subgraphs together to generate efficient code. DLVM [24] is similar to TensorFlow XLA, which focuses on the high-level more, but it promotes using linear algebra instead of the computation graph to express the higher-level of the neural network. As they pay less attention to the hardware level, it needs significant engineering effort for each hardware and operation combination. And they all not support Sunway architectures.

TVM [4] proposes the end-to-end compiler for neural networks and now supports various hardware. Recent works such as Glow [19], Tensor Comprehensions [21] and nGraph [8] can all be classified into this category. Glow emphasis its two-phase strongly-typed intermediate representation and nGraph pay more attention to how to simplify the connection between neural network frameworks and hardware. Tensor Comprehensions provides a language similar to math to describe the neural network and supports optimizing the computational kernel according to the parameter of neural networks in JIT mechanism. But they all not support the Sunway architecture.

There are also research works that are extending TVM such as VTA [18] and AutoTVM [5]. VTA is proposed to build an end-to-end deep learning stack from the high-level deep learning framework to the actual hardware design and implementation directly. And AutoTVM is designed to automatically optimize the tensor operator regarding the specific input shape and layout of each neural network layer.

7.2 Performance Optimization on Sunway

As a supercomputer consisting of massive Sunway many-cores processors, Sunway TaihuLight achieved the peak performance of 125PFlops and ranked the first place in Top500 from 2016 to 2018. There are a lot of optimization works targeting the architecture features on Sunway, which are valuable for our work to generate high performance code.

For applications, Dynamic Model [25] and Earthquake Simulation [10] both win the Gordon Bell Prize of ACM. Dynamic Model simulates the coarsening dynamics accurately at unprecedented spatial and time scales, whereas Earthquake Simulation enables the simulation of the Tangshan earthquake as 18-Hz scenario with 8-meter resolution. For algorithms, there are plenty of algorithms optimized on Sunway such as BFS [16], SpMV [17] and SpTRSV [15]. BFS [16] is an essential algorithm in calculating the shortest route and the maximum flow problem, and the optimization on Sunway achieves 23,755.7 giga-traversed edges per second. Sparse computation such as SpMV is one of the important computational kernels in scientific applications. The implementation of SpMV [17] on Sunway achieves 15.5× speedup on average over 18 representative datasets. Multi-role SpTRSV [23] assigns different processing roles to the CPEs that achieves 5.14× speedup on average over 12 representative datasets. There are also two related works regarding the neural network on Sunway. *swDNN* [9] is a neural network library customized for Sunway with tremendous engineering efforts.

swCaffe [14] proposes a deep learning framework specialized for distributed training on Sunway.

To the best of our knowledge, there is no existing work on the end-to-end deep learning compiler that exploits the architecture advantage of Sunway processor.

8 CONCLUSION AND FUTURE WORK

To improve the productivity and performance of deep neural network, new deep learning frameworks and hardware devices are emerging rapidly. The end-to-end deep learning compiler provides an efficient way to deploy the deep learning applications to various hardware devices without heavy engineering efforts. To provide such a compiler on Sunway architecture, this paper proposes *swTVM* that implements AOT code generation to address the unique compilation environment on Sunway such as cross-compilation and different compile targets for MPE and CPE. In addition, *swTVM* adopts several architecture features on Sunway to improve the performance of the generated code. Specifically, a DMA control interface is proposed to better manipulate the data access of the tensor. Moreover, a LDM management mechanism is designed to buffer data in LDM in order to reduce the memory access latency. Finally, a DMA auto-insertion algorithm is proposed to identify the locations for inserting DMA instructions automatically with improved data re-accessibility. The experimental results of AlexNet and VGG-19 show that the generated code by *swTVM* achieves 6.71× and 2.45× speedup over hand-optimized OpenACC implementations on convolution and fully connected layer respectively. For the future work, we would like to exploit other optimization techniques such as register communication and instruction re-ordering to further improve the performance of the generated code on Sunway.

REFERENCES

[1] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, et al. 2016. Tensorflow: a system for large-scale machine learning.. In *OSDI*, Vol. 16. 265–283.

[2] Mariusz Bojarski, Davide Del Testa, Daniel Dworakowski, Bernhard Firner, Beat Flepp, Prasoon Goyal, Lawrence D Jackel, Mathew Monfort, Urs Muller, Jiakai Zhang, et al. 2016. End to end learning for self-driving cars. *arXiv preprint arXiv:1604.07316* (2016).

[3] Tianqi Chen, Mu Li, Yutian Li, Min Lin, Naiyan Wang, Minjie Wang, Tianjun Xiao, Bing Xu, Chiyuan Zhang, and Zheng Zhang. 2015. Mxnet: A flexible and efficient machine learning library for heterogeneous distributed systems. *arXiv preprint arXiv:1512.01274* (2015).

[4] Tianqi Chen, Thierry Moreau, Ziheng Jiang, Lianmin Zheng, Eddie Yan, Haichen Shen, Meghan Cowan, Leyuan Wang, Yuwei Hu, Luis Ceze, et al. 2018. TVM: An Automated End-to-End Optimizing Compiler for Deep Learning. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*. 578–594.

[5] Tianqi Chen, Lianmin Zheng, Eddie Yan, Ziheng Jiang, Thierry Moreau, Luis Ceze, Carlos Guestrin, and Arvind Krishnamurthy. 2018. Learning to Optimize Tensor Programs. *arXiv preprint arXiv:1805.08166* (2018).

[6] Sharan Chetlur, Cliff Woolley, Philippe Vandermersch, Jonathan Cohen, John Tran, Bryan Catanzaro, and Evan Shelhamer. 2014. cudnn: Efficient primitives for deep learning. *arXiv preprint arXiv:1410.0759* (2014).

[7] Kyunghyun Cho, Bart Van Merriënboer, Caglar Gulcehre, Dzmitry Bahdanau, Fethi Bougares, Holger Schwenk, and Yoshua Bengio. 2014. Learning phrase representations using RNN encoder-decoder for statistical machine translation. *arXiv preprint arXiv:1406.1078* (2014).

[8] Scott Cyphers, Arjun K Bansal, Anahita Bhiwandiwalla, Jayaram Bobba, Matthew Brookhart, Avijit Chakraborty, Will Constable, Christian Convey, Leona Cook, Omar Kanawi, et al. 2018. Intel nGraph: An Intermediate Representation, Compiler, and Executor for Deep Learning. *arXiv preprint arXiv:1801.08058* (2018).

[9] Jiarui Fang, Haohuan Fu, Wenlai Zhao, Bingwei Chen, Weijie Zheng, and Guangwen Yang. 2017. swDNN: A library for accelerating deep learning applications on sunway taihulight. In *Parallel and Distributed Processing Symposium (IPDPS), 2017 IEEE International*. IEEE, 615–624.

[10] Haohuan Fu, Conghui He, Bingwei Chen, Zekun Yin, Zhenguo Zhang, Wenqiang Zhang, Tingjian Zhang, Wei Xue, Weiguo Liu, Wanwang Yin, et al. 2017. 18.9-Pflops nonlinear earthquake simulation on sunway taihulight: Enabling depiction of 18-Hz and 8-meter scenarios. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. ACM, 2.

[11] Yangqing Jia, Evan Shelhamer, Jeff Donahue, Sergey Karayev, Jonathan Long, Ross Girshick, Sergio Guadarrama, and Trevor Darrell. 2014. Caffe: Convolutional architecture for fast feature embedding. In *Proceedings of the 22nd ACM international conference on Multimedia*. ACM, 675–678.

[12] Nikhil Ketkar. 2017. Introduction to pytorch. In *Deep Learning with Python*. Springer, 195–208.

[13] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. 2012. Imagenet classification with deep convolutional neural networks. In *Advances in neural information processing systems*. 1097–1105.

[14] Liandeng Li, Jiarui Fang, Haohuan Fu, Jinlei Jiang, Wenlai Zhao, Conghui He, Xin You, and Guangwen Yang. 2018. swCaffe: A Parallel Framework for Accelerating Deep Learning Applications on Sunway TaihuLight. In *2018 IEEE International Conference on Cluster Computing (CLUSTER)*. IEEE, 413–422.

[15] Mingzhen Li, Yi Liu, Hailong Yang, Zhongzhi Luan, and Depei Qian. 2018. Multi-role SpTRSV on Sunway Many-Core Architecture. In *2018 IEEE 20th International Conference on High Performance Computing and Communications; IEEE 16th International Conference on Smart City; IEEE 4th International Conference on Data Science and Systems (HPCC/SmartCity/DSS)*. IEEE, 594–601.

[16] Heng Lin, Xiongchao Tang, Bowen Yu, Youwei Zhuo, Wenguang Chen, Jidong Zhai, Wanwang Yin, and Weimin Zheng. 2017. Scalable graph traversal on sunway taihulight with ten million cores. In *Parallel and Distributed Processing Symposium (IPDPS), 2017 IEEE International*. IEEE, 635–645.

[17] Changxi Liu, Biwei Xie, Xin Liu, Wei Xue, Hailong Yang, and Xu Liu. 2018. Towards Efficient SpMV on Sunway Manycore Architectures. In *Proceedings of the 2018 International Conference on Supercomputing*. ACM, 363–373.

[18] Thierry Moreau, Tianqi Chen, Ziheng Jiang, Luis Ceze, Carlos Guestrin, and Arvind Krishnamurthy. 2018. VTA: An Open Hardware-Software Stack for Deep Learning. *arXiv preprint arXiv:1807.04188* (2018).

[19] Nadav Rotem, Jordan Fix, Saleem Abdulrasool, Summer Deng, Roman Dzhabarov, James Hegeman, Roman Levenstein, Bert Maher, Satish Nadathur, Jakob Olesen, et al. 2018. Glow: Graph Lowering Compiler Techniques for Neural Networks. *arXiv preprint arXiv:1805.00907* (2018).

[20] Karen Simonyan and Andrew Zisserman. 2014. Very deep convolutional networks for large-scale image recognition. *arXiv preprint arXiv:1409.1556* (2014).

[21] Nicolas Vasilache, Oleksandr Zinenko, Theodoros Theodoridis, Priya Goyal, Zachary DeVito, William S Moses, Sven Verdoolaege, Andrew Adams, and Albert Cohen. 2018. Tensor Comprehensions: Framework-Agnostic High-Performance Machine Learning Abstractions. *arXiv preprint arXiv:1802.04730* (2018).

[22] Endong Wang, Qing Zhang, Bo Shen, Guangyong Zhang, Xiaowei Lu, Qing Wu, and Yajuan Wang. 2014. Intel math kernel library. In *High-Performance Computing on the Intel® Xeon PhiâĎć*. Springer, 167–188.

[23] Xinliang Wang, Wei Xue, Weifeng Liu, and Li Wu. 2018. swSpTRSV: a fast sparse triangular solve with sparse level tile layout on sunway architectures. In *Proceedings of the 23rd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. ACM, 338–353.

[24] Richard Wei, Lane Schwartz, and Vikram Adve. 2017. DLVM: A modern compiler infrastructure for deep learning systems. *arXiv preprint arXiv:1711.03016* (2017).

[25] Jian Zhang, Chunbao Zhou, Yangang Wang, Lili Ju, Qiang Du, Xuebin Chi, Dongsheng Xu, Dexun Chen, Yong Liu, and Zhao Liu. 2016. Extreme-scale phase field simulations of coarsening dynamics on the sunway taihulight supercomputer. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE Press, 4.

[26] Kaipeng Zhang, Zhanpeng Zhang, Zhifeng Li, and Yu Qiao. 2016. Joint face detection and alignment using multitask cascaded convolutional networks. *IEEE Signal Processing Letters* 23, 10 (2016), 1499–1503.