Research Article

Simulating Neural Network Processors

Jian Hu, Xianlong Zhang, and Xiaohua Shi

College of Software, Beihang University, Beijing, China

Correspondence should be addressed to Xiaohua Shi; xhshi@buaa.edu.cn

Received 6 January 2022; Revised 21 January 2022; Accepted 26 January 2022; Published 23 February 2022

Academic Editor: Mohammad Farukh Hashmi

Copyright © 2022 Jian Hu et al. This is an open access article distributed under the Creative Commons Attribution License, which permits unrestricted use, distribution, and reproduction in any medium, provided the original work is properly cited.

Deep learning has achieved competing results compared with human beings in many fields. Traditionally, deep learning networks are executed on CPUs and GPUs. In recent years, more and more neural network accelerators have been introduced in both academia and industry to improve the performance and energy efficiency for deep learning networks. In this paper, we introduce a flexible and configurable functional NN accelerator simulator, which could be configured to simulate u-architectures for different NN accelerators. The extensible and configurable simulator is helpful for system-level exploration of u-architecture, as well as operator optimization algorithm developments. The simulator is a functional simulator that simulates the latencies of calculation and memory access and the concurrent process between modules, and it gives the number of program execution cycles after the simulation is completed. We also integrated the simulator into the TVM compilation stack as an optional backend. Users can use TVM to write operators and execute them on the simulator.

1. Introduction

Deep learning has been applied to image recognition, object detection, speech recognition, and other fields. In some fields, deep learning has even achieved competing results compared to humans. Traditionally, CPUs and GPUs are widely used to execute neural networks (NNs), but more and more hardware accelerators have been introduced to improve the performance and energy efficiency of NN computing.

Tensor Processing Unit (TPU) [1] is a custom ASIC which was deployed in Google's data center in 2015 and used to speed up the inference phase of NNs. TPU uses a 8-bit integer systolic matrix multiplier to do inference, and it contains on-chip buffers with a total capacity of nearly 30 MiB. TPU supports MLP, RNN, and CNN, and its inference performance is 15X–30X times faster than the NVIDIA K80 GPU and Intel E5-2699 CPU. The DianNao series, including DianNao [2], DaDianNao, ShiDianNao, and PuDianNao, were introduced in 2014 by ICT, CAS. DaDianNao [3] is a multi-chip deep learning inference and training architecture and supports both 16-bit and 32-bit fixed point computing. DaDianNao supports convolution, pooling, classifier, and LRN layers, and when using a 64-node architecture, it

achieves more than 2000X accelerations in convolution computation compared to GPU baselines. ShiDianNao [4] focuses on accelerating convolution operations in embedded applications and supports pooling, classification, and normalization layers as well. ShiDianNao uses inter-PE data propagation to reduce memory access in convolution, which makes it highly energy efficient. PuDianNao [5] is a polyvalent ML accelerator and focuses on accelerating ML techniques besides DNN (deep neural network), such as SVM, K-means, classification tree, and so on. Cambricon [6] is the first instruction set architecture (ISA) for deep learning. Inspired by RISC ISA principle, rather than mapping each NN layer to a single instruction, Cambricon ISA decomposes NN layers into basic computations and maps those computations to simple instructions. By doing so, users can further assemble new NN layers from those basic computations, which makes Cambricon ISA more flexible than its predecessors. They also implemented a prototype accelerator of Cambricon ISA, which achieved the same level of performance as DaDianNao in the experiments.

Some studies [7–10] have focused on designing ultra-low-power, high power-efficient CNN accelerators for IOT devices. Thinker-II [10] is an energy efficiency

reconfigurable DNN processor for IOT devices that uses binary/ternary weights to do calculations, and it applies three techniques to improve energy efficiency and achieves 19.9TOPS/W power efficiency at a power consumption of 10 mW. There are also DNN processors that use in-memory processing technologies, such as Neurocube [11], Prime [12], ISAAC [13], and PipeLayer [14]. They improve energy efficiency by reducing data movement between memory and processing units, and the latter 3 use analog arithmetic for matrix calculations. Eyeriss [15] improves energy efficiency by reducing data movement through reconfigurable computation mapping, with a processing dataflow called row stationary, and it also gates zero neurons to save power. Some NN processors improve performance and energy efficiency by taking advantage of the sparsity of NN models, including Cambricon-X [16], EIF [17], SCNN [18], and so on. Cnvlutin [19] improves performance and energy efficiency by eliminating ineffectual operations in DNN, and it is targeted at lower sparsity between 40 and 50%. Stripes [20] is an NN accelerator that improves performance and energy efficiency by leveraging bit-serial computing units. There are studies which exploit 3D memory for NN accelerator, such as TETRIS [21], and 3D memory allows it to use more area for processing units and simplifies dataflow scheduling. Some studies focus on NN accelerator design and optimization on FPGAs, such as [22–25]. There are some studies that focus on the method of designing NN accelerators, such as Minerva [26], which is a co-design approach to optimize NN accelerator across NN algorithms, architecture, and circuit.

TVM [27] is a deep learning compiler stack, and it provides both graph-level and operator-level optimizations and can target different backends including CPU, GPU, and hardware accelerators. DL networks are first represented in computation graphs in TVM; on this level, TVM can do operator fusion, tensor layout transform, and constant folding optimizations. After graph-level optimizations, operators are lowered to a form represented in TVM's tensor express language and then users can apply TVM's schedule primitives to create a schedule for operators. Schedules in TVM are mapping from tensor expression to actual low-level code to do the computation. For hardware accelerator backends, TVM provides tensorize schedule primitive, which can pattern match an unit of computation and replace it with accelerator's instruction.

VTA [28] is a programmable deep learning accelerator co-designed with TVM and is integrated into TVM. VTA has a two-level ISA: the first is task-level parallel memory transfer and compute ISA and the second is a microcoded ISA which operates basic vector and matrix computations.

Hardware NN accelerator architecture is in a rapid development phase, so a parametric simulator that can be compatible with many NN accelerator architectures' characteristics is of great practical value to chip designers and compiler writers. It can be used to help with system-level exploration of chip architecture or verify the effectiveness of optimizations of compilers or evaluate the scheduling strategy of NN operators' implementations.

The contributions of this paper are as follows:

- (a) Different NN accelerators have different ISA, memory hierarchy, and execution unit (types and sizes), so we designed and implemented a flexible and configurable NN accelerator simulator that is easy to extend and allows parameters of the simulated architecture to be modified by modifying configuration file.
- (b) The simulator is a functional simulator that simulates the latencies of calculation and memory access and the concurrent process between modules, and it gives the number of program execution cycles after the simulation is completed.
- (c) We also integrated the simulator into the TVM compilation stack as a backend, and the users can use TVM to write and generate operators and execute them on the simulator.

The rest of this paper is organized as follows. Section 2 introduces the architecture and ISA of the accelerator. Section 3 presents the software implementation of the simulator. Section 4 introduces the code generation system. Section 5 evaluates the simulator by different configurations.

2. Accelerator Architecture and ISA

In this section, we will present the architecture and instruction set architecture we designed and implemented in the simulator. The ISA is a two-level ISA: the first level is an RISC-like ISA, which contains instructions such as integer scalar calculation, branching, and so on. The second level is an ISA like the microcoded ISA of VTA, which is responsible for tensor calculation and tensor data transfer, such as GEMM instructions. Multiple microcoded ISA instructions form a microcoded kernel executed on the corresponding tensor pipeline. There is an extension instruction in the first-level ISA to launch a microcoded kernel on the corresponding tensor pipeline. In addition, the tensor pipelines employ the decoupled access-execute (DAE) architecture, so the first-level ISA contains two more instructions (dependency push/pop instructions) for explicit synchronization between tensor pipelines.

2.1. Accelerator Architecture. Figure 1 shows the top-level block figure of this architecture. As illustrated in this figure, the architecture contains several stages to execute first-level ISA instructions: fetching, decoding, dispatching and register reading, and execution and writing back. The integer scalars, which represent loop variables or addresses, are stored in register files and scalar memory. Similar to a typical RISC architecture, only the load/store instructions can access the scalar memory. A first-level ISA instruction will be fetched, decoded, and then injected into the in-order dispatch queue. When one instruction reaches the head of dispatch queue, without dependency or anti-dependency to others, it will be dispatched to the corresponding execution unit. Branch instructions and integer scalar arithmetical and logical instructions will be sent to ALU, while scalar load/store instructions will be sent to load-store-unit (LSU).

Other instructions for tensor pipeline synchronization or launching microcoded kernel will be sent to the corresponding tensor pipeline. There are FIFO queues between the dispatch queue and the execution unit or tensor pipeline, which are not shown in Figure 1. After execution, the results will be written to the register file. It should be noted that after one branch instruction has been dispatched, the dispatch process must be stalled until the branch result is known; because there is no re-ordering buffer in this architecture, the results will be committed directly and cannot be canceled.

In addition to the pipeline stages that execute the first-level ISA instructions, there are several tensor pipelines that execute the microcoded ISA instructions. Each tensor pipeline contains a pipeline controller, as well as an execution unit to perform actual works, and the execution unit may contain multiple pipeline stages inside. The pipeline controller is responsible for synchronizations between pipelines, as well as launching microcoded kernels (decoding and issuing microcoded ISA instructions in that kernel). Tasks running on different tensor pipelines are synchronized with the dependency pop/push instructions, to avoid data hazards. When a dependency push instruction is issued to a pipeline controller, the controller pushes a token to the corresponding dependency token queue or waits for an empty slot to be available. In the case of the dependency pop instruction, the controller will either pop a token from the corresponding dependency token queue or wait for a token to be available. A microcoded kernel consists of several microcoded ISA instructions and is stored in the corresponding pipeline controllers' internal storage. A launch kernel instruction contains two register operands, representing the extent of two nested loops. When a launch kernel instruction is issued to the pipeline controller, the controller uses the two operands as the extent to perform a two-level loop, and in the loop body, it iteratively reads each microcoded ISA instructions of the kernel, decoding and then issuing it into the execution unit. The procedure of launching a microcoded kernel and decoding and issuing each instruction is shown in Algorithm 1.

Tensors, which represent data or weights, are stored in addressable on-chip scratchpad memories. In our simulator, we make the number of scratchpad memory configurable, and each tensor execution unit can access each memory, which makes the simulator more flexible. In our simulator, the capacity, bank width, bank number, read/write latencies, and type (one-port, or two-port) of each scratchpad memory are configurable. The simulator models the bank conflict between scratchpad memory accesses as well.

Current simulator implementation contains $3 + N$ tensor pipelines and can be easily extended.

The first pipeline is the matrix computing pipeline, which executes the microcoded ISA instructions of GEMM. The execution unit (later referred to as MAC) of matrix computing pipeline is divided into 4 pipeline stages: read, multiply, reduce, and accumulate/write back. In the read stage, tensor operands are read from scratchpad memories. The multiply stage contains many multipliers that perform element-wise multiplication on broadcasted inputs. The

reduce stage contains many adder trees that reduce sum the output of the multiply stage and get multiple dot products. In the last stage, the results are accumulated into partial sums stored in one scratchpad memory (later referred to as accumulation buffer) or written back directly to one scratchpad memory. The simulator allows the input/output tensor size of the matrix computation instructions to be variable, with more flexibility and configurability. These sizes are embedded into the microcoded ISA instructions.

The second tensor pipeline is a vector computation pipeline, which performs vector activation functions, vector element-wise arithmetical and relational calculations, vector-immediate/scalar arithmetical and relational calculations, etc. The execution unit of this pipeline is divided into three pipeline stages: read, calculate, and write. As with matrix computing pipeline, we allow the input/output tensor sizes to be variable and embed the sizes into the instructions. Unlike matrix computing pipelines, vector computing pipelines cannot perform accumulate operation.

The third tensor pipeline is a memory transfer pipeline that is responsible for the data transfer between the scratchpad memory and the off-chip memory. Each memory transfer instruction can perform 2D memory transfer, making it easier to do data tiling. The remaining N tensor pipelines are also memory transfer pipelines, which are responsible for data transfer between scratchpad memories. N is equal to the number of scratchpad memories, and the i -th one is responsible for copying data from other scratchpad memory to the i -th scratchpad memory. Although there are $N * N$ combinations of scratchpad memory pairs, we think N such pipelines are sufficient for modeling an actual architecture.

2.2. Accelerator ISA. Table 1 shows an overview of the instructions. As mentioned earlier, the ISA is a two-level ISA:

- (1) The first level is a RISC-like ISA, containing integer scalar calculation, branching and load/store instructions, and several extension instructions: instructions for synchronization between tensor pipelines, instructions for launching microcoded kernel, and instructions for assigning value to pipeline controller's local registers. The operands of these instructions are registers or integer numbers. The load/store instructions can be used to spill registers into a small scalar memory or load parameters set by the host from the scalar memory. This level of ISA can be used to implement the control flow, which also makes it possible to influence the control flow with the value of the tensor data, which is necessary for operators such as ROI pooling to be executed without copying data back to host.
- (2) The second level is microcoded ISA, which performs vector and matrix calculations as well as tensor memory transfers. Instructions in this ISA level cannot access the general register file. Their operands can be an integer immediate value, an immediate

FIGURE 1: The top-level block figure of the u-architecture.

```

Input: Two loop extents, ext0, ext1; the ID of the kernel to be launched, kId
(1) kernel = get the kernel with ID kId
(2) for var0 = 0; var0 < ext0; var0 ++ do
(3)   for var1 = 0; var1 < ext1; var1 ++ do
(4)   for all insn in instructions of kernel do
(5)     while there is no free slot in output queue do
(6)       end this cycle
(7)     end while
(8)     write opcode of insn to output
(9)     for all arg in arguments of insn do
(10)    if arg is immediate argument then
(11)      argval = arg
(12)    else if arg is composite argument then
(13)      argval = arg.coef0 * var0 + arg.coef1 * var1 + arg.addend + localregs[arg.reg]
(14)    end if
(15)    write argval to output
(16)  end for
(17) end this cycle
(18) end for
(19) end for
(20) end for

```

ALGORITHM 1: The procedure of launching one microcoded kernel.

value of tensor data types, or a composite operand (consisting of two coefficients, an addend, and one pipeline controller local register and is in the form of

$\{coef0, coef1, addend, local_reg\}$). These operands are often used to represent the addresses of tensors, as well as strides or other parameters.

Multiple microcoded ISA instructions form a microcoded kernel and will be stored in the pipeline controller. When launched by the pipeline controller, the instructions in the microcoded kernel are fetched and decoded by the controller and sent to the corresponding tensor pipeline execution unit for execution.

Figure 2 presents the program fragment for fully connected layer, and Figure 3 shows the source IR right before codegen. For brevity, we omit some instructions for register assignment and pipeline synchronization. Figure 2(a) shows instructions of the 1st level ISA, which performs loops, pipeline synchronization, and kernel launching operations, and Figure 2(b) shows the microcoded kernels composed of the 2nd level ISA. As we can see, the 1st level ISA instructions contain only one loop (namely, the loop online 6 in original IR), and the rest of the loops are performed by the pipeline controllers when executing the LaunchKernel instructions (see Algorithm 1). The 3rd and 4th parameters of each LaunchKernel instruction represent the two loop extents.

3. Simulator Implementations

The goal of our simulator is to be a functional simulator, which means the simulator could simulate the concurrent processes between modules and pipelines, the latencies of memory accesses, instruction executions, and the bank conflicts of memory accesses. After completing the simulation of the program, it needs to give the number of cycles it takes to execute the program. In addition, the simulator must be sufficiently configurable. By modifying the configuration file or predefined configuration constants in the front-end code, users can modify the parameters of the simulated architecture, including

- (1) The amount of scratchpad memories, the capacity, the number of banks, the bit-width, the read and write latency, and the type (one-port or two-port) of each scratchpad memory.
- (2) The execution latency for each instruction at each execution unit or pipeline stage.
- (3) Depth of various types of queues, such as queues from dispatch queues to execution units or pipeline controllers.
- (4) Data types of tensors. The simulator supports a variety of data types and mixed precision calculations. The data types of inputs and outputs of an instruction are encoded in the instruction.
- (5) The input and output tensor sizes of vector/matrix calculation instructions. These sizes are encoded in the instruction. This is equivalent to allowing the size of MAC and vector execution unit to be configurable.

Given the above requirements, we chose to develop simulator based on SystemC [29], which is a C++ class library providing a discrete event simulation interface. It contains a set of classes and macros, enabling users to

simulate concurrent processes, which is important for hardware simulation. It also provides a notion of time into C++, enabling event sequencing. SystemC also provides data types for hardware modeling, such as 4-value logic vector, but users can still use C++ types. Figure 4 shows the software architecture diagram of the simulator, which can be divided into three parts: modules, channels, and common functions. Each one in the Modules part is a C++ class that inherits from `sc_module` (a base class in SystemC) and usually corresponds to a module/stage in the accelerator architecture. For example, the IFetch, IDecode, IDispatch, and ALU in this figure correspond to fetch, decode, dispatch, and ALU in Figure 1. A module can have several submodules, for example, Matrix Unit has four submodules, each representing one of the four pipeline stages. The read stage module further contains one Scratchpad Reader submodule for scratchpad memory read access. There are also inheritance hierarchies between module classes, for example, several different but similar pipeline controllers are required in the implementation, and inheritance mechanisms can help with code reuse. Each module has one or more processes (essentially member functions) which implement the function logic of the module. Modules interact with each other through channels, including the built-in channels of SystemC (such as `sc_signal`), and our custom channels, such as the Dependency Hub in the figure.

A module calls the methods of channels to write or read data, which transfer data between Modules or other functions. The Dependency Hub is a centralized dependency queue channel, through which each tensor pipeline sends dependency tokens to another pipeline and also gets tokens from others. The Scratchpad Memory Hub is a centralized channel that manages all scratchpad memories, and through it, each module can easily accesses any scratchpad memory. Using centralized channels makes it easier to extend new pipelines, as well as configure the memory hierarchy of simulated architecture, etc. SRAM channels are a set of channels used to model scratchpad memory bank, all of which implement the same channel interface with different latencies and mutual-exclusive access modeling implementations. Classes in the Common part implement a variety of basic functions. The Memory class, for example, represents a memory. It implements methods of reading data, writing data, and memset, but it has nothing to do with latency. BitPacker and BitPackerImpl are classes that represent the tensor data transferred between the modules. BitPacker is an abstract class that defines a set of virtual functions to perform type cast, element access, arithmetical and logical operations, and other operations. BitPacker manages the underlying data, as well as the size information. BitPackerImpl is a template class inherited from BitPacker, which represents tensors of a specific data type and implements the virtual functions defined by BitPacker. Further, we can extend data types to those not supported by C++, such as low bit and fixed-point numbers, by implementing the wrapper class of other data types and using it as a template parameter for BitPackerImpl.

TABLE 1: An overview of instructions.

Type	Instructions	Operands
1 st level	Control Scalar Load/store	Branch, jump Integer scalar arithmetical, relational, logical Integer scalar load and store
2 nd level	Vector Matrix Data transfer	Vector activation functions, vector binary arithmetical and relational, vector-immediate/scalar arithmetical and relational, etc. GEMM Tensor DMA load/store, tensor copy, im2col
		Immediate, register Integer immediate, tensor immediate, composite operand

# \$g5: bias address #	#0:
\$g4: output address #	DMABufLoad {0:0:0:1}, 0,
\$g6: weight address #	{0:0:0x50000000:0}, 0, 128, 1
\$g7: data address	#1:
SetPipelineReg 1, 1,	Memset {1024:128:0x30000000:0}, 8,
\$g5 # \$g5=1, \$g8=8, \$g10=4	8, 16, 0.0, 1
LaunchKernel 1, #0, \$g5, \$g5	#2:
LaunchKernel 7, #1, \$g8, \$g8	DMABufLoad {8192:8:0:1}, 1024,
# pipeline sync omitted	{256:64:0x10000000:0}, 8, 8, 8
DependPush 2, 1	#3:
% # \$g9 is loop var	DMABufLoad
AddIU \$g9, \$zero, 0	#4:
.LBB0_1:	GEMM {1024:128:0x30000000:0},
DependPop 2, 1	16, {256:0:0x10000000:0}, 8,
SHLI \$g11, \$g9, 5	{0:256:0x20000000:0}, 8,
AddU \$g12, \$g7, \$g11	GEMM {1024:128:0x30000000:0},
SetPipelineReg 1, 1, \$g12	16, {256:0:0x10000040:0}, 8,
LaunchKernel 1, #2, \$g8, \$g10	{0:256:0x20000040:0}, 8,
AddU \$g12, \$g6, \$g11	GEMM
SetPipelineReg 1, 1, \$g12	GEMM.....
LaunchKernel 1, #3, \$g8, \$g10	#5:
DependPush 1, 2	MAddV {512:64:0x40000000:0}, 8,
DependPop 1, 2	{1024:128:0x30000000:0}, 16,
LaunchKernel 2, #4, \$g8, \$g8	{0:16:0x50000000:0}, 8, 8, 2
DependPush 2, 1	#6:
AddIU \$g9, \$g9, 1	DMABufStore {0:0:0:1}, 0,
SLTIU \$g11, \$g9, 32	{0:0:0x40000000:0}, 0, 4096, 1)
BNEZ \$g11, .LBB0_1	
# pipeline sync omitted	
LaunchKernel 3, #5, \$g8, \$g8	
# pipeline sync omitted	
SetPipelineReg 1, 1, \$g4	
LaunchKernel 1, #6, \$g5, \$g5)	

Assembly of 1st-level code.

Microcoded kernels.

FIGURE 2: Code fragment for fully connected layer. (a) Assembly of 1st level code. (b) Microcoded kernels.

```

NPU.DMABufLoad(bias, 0, 0, (0 + 0x50000000), 0, 128, 1)
for (xo, 0, 8) {
  for (yo, 0, 8) {
    NPU.Memset((((xo*8) + yo)*128) + 0x30000000), 8, 8, 16, 0.0, 1)
  }
  for (ko, 0, 32) {
    for (io, 0, 8) {
      for (il, 0, 4) {
        NPU.DMABufLoad(a, (((io*1024) + il) + (ko*4))*8, 1024,
          (((io*4) + il)*64) + 0x10000000), 8, 8, 8)
      }
      for (io, 0, 8) {
        for (il, 0, 4) {
          NPU.DMABufLoad(b, ..... )
        }
        for (xo, 0, 8) {
          for (yo, 0, 8) {
            for (ki, 0, 4) {
              NPU.GEMM((((xo*8) + yo)*128) + 0x30000000), 16,
                (((xo*4) + ki)*64) + 0x10000000, 8,
                (((yo*4) + ki)*64) + 0x20000000, 8, ..... )
            }
          }
          for (no, 0, 8) {
            for (oco, 0, 8) {
              NPU.MAddV((((no*8) + oco)*64) + 0x40000000), 8,
                (((no*8) + oco)*128) + 0x30000000, 16,
                ((oco*16) + 0x50000000), 8, 8, 2)
            }
          }
          NPU.DMABufStore(out_host, 0, 0, (0 + 0x40000000), 0, 4096, 1)
        }
      }
    }
  }
}

```

FIGURE 3: IR for fully connected layer.

4. Codegen System

We integrate our backend into TVM by using TVM's tensorize primitive and some custom IR passes and replacing the original code segment with intrinsic calls corresponding to microcoded instructions of our architecture, e.g., GEMM. After this step, IRs become the form in Figure 3 and continue with the codegen operation. Our architecture uses a two-level ISA, so we first need to split the IR into 2 parts: the microcoded kernels consisting of the 2nd level ISA instructions and the IR consisting of the 1st level ISA instructions. Microcoded kernels can be assembled and loaded by the simulator, while the 1st level ISA IR is further processed by code generator.

Since the first-level ISA is a register-based ISA, we need a code generator to do instruction selection, instruction scheduling, and register allocation. We use the target-independent code generator of LLVM to do this work. The main job here is to write description files for our architecture's registers, register classes, instructions of the first-level ISA, and the selection patterns. We

register LLVM intrinsic functions for instructions without corresponding LLVM instructions, including instructions launching the microcoded kernel and the dependency pop/push instructions. When LLVM IRs are generated from TVM IRs, these intrinsic calls will be converted to the appropriate LLVM intrinsic function calls.

We use an example to illustrate how to split the IR, for example, given the IR segment in Figure 3 for fully connected layer, the code generator will convert IR codes from lines 16 to 22 into a microcoded kernel. It first verifies whether the step lengths of the outermost two loops are equal to 1 and extracts the loop variables *xo* and *yo*. It also verifies whether inner loops have constant loop domains. Then, it handles the GEMM call from lines 19 to 21 and processes each argument according to the instruction template: for an immediate parameter, it verifies the type and value range of the arguments; for a composite parameter, it tries to represent the argument in the form of $coef0 * xo + coef1 * yo + addend + free_expr$, where *coef0* and *coef1* are integers, *addend* contains only loop variables of inner loops, and *free_expr* contains only free variables that are defined outside the scope. Then, the parameter is converted to a composite parameter like $\{coef0, coef1, addend, reg(free_expr)\}$. After completing the previous steps, the code generator will unroll all inner loops, generate the microcoded kernels, and replace the original IR node with a launch kernel instruction along with instructions to set corresponding local registers. Figure 5 contains the IR codes after transformation, with only one loop in it. Figure 2(a) shows the assembly codes of the first-level IRs, which are generated from the transformed IRs in Figure 5. In this example, the 1st, 3rd, and 5th parameters of the GEMM instruction are composite parameters, which represent the input/output addresses. The loop online 18 has been unrolled, resulting in 4 GEMM microcoded instructions in the generated kernel. Figure 2(b) shows the extracted microcoded kernels, in which the one labeled as #4 is extracted from lines 16 to 21 of the original IRs, which contain 4 GEMM instructions.

5. Experiments

5.1. Functions of the Simulator. In this section, we use the matrix multiplication example to demonstrate functions of the simulator, showing that it simulates the bank conflicts, memory access latency, and the concurrency between pipelines. For example, we will compute $Y = XW^T$, where *X*: (128, 1024), *W*: (128, 1024). The configurations of test architecture are as follows:

- (1) The shape of MAC is $8 \times 8 \times 8$. That means each GEMM instruction reads two 8×8 tensors and generates one 8×8 result. The throughput of the GEMM instruction is set to 1 per cycle (1/cycle).
- (2) Using 2 on-chip scratchpad memories to store input data and weight. The numbers of banks are both 8, and both have capacity of 32 kB. The width of each bank is 8 bytes, the latencies are 1 cycle, and are all

FIGURE 4: The software architecture of the simulator.

```

llvm.NPU.SetPipelineReg(1, 1, bias)
llvm.NPU.LaunchKernel(1, 0, 1, 1)
llvm.NPU.LaunchKernel(7, 1, 8, 8)
for (ko.outer, 0, 32) {
    llvm.NPU.SetPipelineReg(1, 1, a + (ko.outer*32)) 6 llvm.NPU.LaunchKernel(1, 2, 8, 4)
    llvm.NPU.SetPipelineReg(1, 1, b + (ko.outer*32)) 8 llvm.NPU.LaunchKernel(1, 3, 8, 4)
    llvm.NPU.LaunchKernel(2, 4, 8, 8)
}
llvm.NPU.LaunchKernel(3, 5, 8, 8)
llvm.NPU.SetPipelineReg(1, 1, out_host)
llvm.NPU.LaunchKernel(1, 6, 1, 1)

```

FIGURE 5: The IR after extracting microcoded kernel.

configured as two-port memory. The accumulation buffer has a capacity of 32 kB. It is configured so that its latency does not become a bottleneck. It has two ports as well.

- (3) The input data type of the GEMM instruction is int8, and the output data type is int16.

The simple method of completing the matrix multiplication is to decompose the whole calculation into multiple 8×8 matrix multiplications, loading two submatrices of shape $8 \times (8 \times K)$ from both the X and W matrices at a time, where K should not be too small to avoid the overload of integer calculation, branch instructions, pipeline synchronization, and launching micro-coded kernel. Then, we use the GEMM instruction to complete the partial calculation, repeat the operations until the calculation is completed, and save the accumulated final results back into the off-chip memory.

Based on the simple method, we apply several optimization methods to improve execution efficiency. The

simulator will reflect the difference in the execution efficiency after applying optimizations. The optimization methods include the following:

- (1) Tensor tiling: tiling matrices X and W so that the 8×8 submatrices read by GEMM are stored continuously. This optimization helps to reduce or even avoid bank conflicts and improves the scratchpad memory read efficiency.
- (2) Data reuse: for many u-architectures, the cost of loading data from memory could be much higher than performing a single floating-point calculation. The same is true for our simulator, where the cost of loading data from the off-chip memory is much greater than computations. So, we should reuse as much data already loaded into the on-chip scratchpad memory as we can and do more calculations with the same amount of data loaded. We can do this by loading a few more rows of data at once,

for example, we can load a submatrix of $32 \times (8 \times K)$ from both the X and W matrices at a time and then compute a total of $32 \times 32 \times 8 \times K$ MAC operations, with only $2 \times 32 \times 8 \times K$ data being read. It costs only 1/16 data read per MAC operation with the optimization.

- (3) Memory latency hiding: by making the processes of memory operations and calculations as overlapped as possible, we could increase the utilization of the memory and MAC. Since this architecture is an decoupled access-execute (DAE) architecture, we rely on TVM's cthread to do this optimization.

Because the simulator simulates the effects of bank conflicts, memory latencies, and concurrency processes, these optimizations could obviously improve the computational efficiency. Based on the architecture configuration specified earlier, we set the value of K to 4, and the test results for applying these optimization methods are presented in Table 2.

5.2. Configurability of the Simulator. In this section, we use an example to demonstrate the configurability of the simulator and how those configurations affect the performance of the simulation. In this example, we use 3 operators to demonstrate the configurability of the MAC shape, scratchpad memory bandwidth, and other parameters and how these configurations affect the execution performance. The operators include the following:

- (1) A fully connected operator: $Y = XW^T + b$, where X : (128, 1024), W : (1024, 1024).
- (2) A conv2d operator: where input: (1, 16, 16, 256) in NHWC layout, kernel: (3, 3, 256, 256), strides: (1, 1).
- (3) A max pooling operator: where input: (40, 40, 256), ksize: (2, 2), strides: (2, 2).

The top-level block figure of the simulated architecture is shown in Figure 6, in which the scalar pipeline and other control units are ignored. 2 on-chip scratchpad memories (buf0 and buf1) are used to store data and the weight, which provides inputs to the MAC unit, which writes its output to the accumulation buffer. Another 2 smaller scratchpad memories (buf3 and buf4) are used to store the final results and the bias data and are connected to the vector unit. The vector unit can also read data from the accumulation buffer for further calculations and add bias.

The five different configurations are as follows:

- (1) The shape of MAC is $8 \times 8 \times 8$, and the size of the vector unit is 64. The bandwidths and latencies of buf0 and buf1 match exactly what GEMM instructions need, and both have the capacity of 32 kB, and the capacity of the accumulation buffer is also 32 kB, with all tree memories ported. In the configuration, the bandwidth between the off-chip memory and the scratchpad memories is about 1/10 of the bandwidth required by the MAC unit.

TABLE 2: Results of optimizations.

Optimization	#Cycles	MAC utilization (%)
Original	564,749	5.8
Tiling	466,445	7
Data reuse	288,501	11.4
Tiling and data reuse	57,237	57.2
All	36,926	88.7

- (2) The MAC shape is set to $16 \times 16 \times 16$, and the size of vector unit is 256, with other configurations unchanged.
- (3) Increasing the bandwidths of the scratchpad memories, making them to match the requirement of the new MAC and the vector unit shape.
- (4) Quadrupling the capacity of the accumulation buffer, with better data reuse, since larger capacity of the accumulation buffer allows us to load more rows at once. We quadruple the capacity of the buf4 for loading more data at once for the vector unit as well.
- (5) Doubling the bandwidth from the off-chip memory to the on-chip scratchpad memories.

For each configuration, the total cycles and the utilization of the MAC/vector unit of the 3 operators are shown in Table 3. As shown in the table, for the fully connected and the conv2d operators, simply increasing the shape of MAC can hardly improve performance but leads to a low MAC utilization. Increasing the scratchpad memory bandwidth has also not resulted in significant performance improvements. It was not until the accumulation buffer capacity and the off-chip to on-chip bandwidth are increased that significant performance improvements were made. Even so, the MAC utilization is not objective, possibly because the computational load of the matrix multiplication in this example is too small relative to the new MAC shape for sufficient data reuse or the memory bandwidth is still not large enough. As for the max pooling operator, because it is a memory bound operator (no data reuse possible), the vector unit utilization is always low, and the performance almost entirely depends on the memory bandwidth.

5.3. Mimicking Other u-Architectures. Finally, we use an example to demonstrate the ability of the simulator to mimic a particular accelerator u-architecture, like the Da Vinci of Huawei [30]. The block figure of the mimicked architecture is shown in Figure 7. The configurations are as follows:

- (1) There is a total of 5 on-chip scratchpad memories. The first (L1 buf) has a capacity of 1 MB and is the only scratchpad memory that is connected to the off-chip memory. The second and third scratchpad memories (L0A and L0B) have a capacity of 64 kB and are connected to the L1 buf, which provides inputs to the MAC unit. The accumulation buffer (L0C buf) has a capacity of 256 kB, which is connected to the MAC unit and the vector unit. The capacity of the fourth scratchpad memory

FIGURE 6: Top-level block figure of the simulated architecture.

TABLE 3: Simulation results of different configurations.

Configuration	Fully connected	Conv2d	Max pooling
	#Cycles MAC utilization (%)	#Cycles MAC utilization (%)	#Cycles MAC utilization (%)
Original	285,379 91.9	315,290 93.5	42,779 15
Larger MAC	266,491 12.3	292,750 12.6	42,779 3.7
Wider scratchpad bandwidth	221,808 14.8	247,330 14.9	38,325 4.2
Larger accumulation buffer	115,187 28.5	127,809 28.8	36,171 4.4
Double bandwidth from off-chip to on-chip	61,923 53.0	68,417 53.9	20,771 7.7

FIGURE 7: Top-level block figure of a mimicked architecture.

(uni buf) is 256 kB and is connected to the l1 buf and vector unit. In the configuration, all memories except L1 buf have bandwidths that meet the requirements of the corresponding execution unit. The bandwidth of L1 buf to L0A or L0B is approximately 1/10 of the bandwidth required by the MAC unit (for both data and weight). Also, the bandwidth between the off-chip memory and the L1 buf is about 1/30 of the bandwidth required by the MAC unit.

- (2) The shape of MAC is $8 \times 8 \times 8$, and the size of vector unit is 64.
- (3) The data type is configured as FP16 + FP32 or int8.

With the above configurations, the memory hierarchy and MAC shape of the simulated micro architecture are similar to the Da Vinci u-architecture. The tensor data type can also be configured to be the same as well. We still use the operators in Section 5.2 as examples to do simulations. The results are shown in Table 4.

TABLE 4: Simulation results of the mimicked u-architecture.

Operator	#Cycles	MAC utilization (%)	Vector unit utilization (%)
Fully connected	377,611	69.4	NA
Conv2d	495,322	59.5	NA
Max pooling	106,980	NA	6

6. Conclusions

In this paper, we present a flexible and configurable functional NN accelerator simulator, which could be configured to simulate u-architectures for different NN accelerators. The simulator is a functional simulator that simulates the latencies of calculation and memory access and the concurrent process between modules, and it gives the number of program execution cycles after the simulation is completed. We also demonstrate how to use the simulator to mimic different microarchitectures like Huawei Ascend and other typical TPUs.

Data Availability

The data used to support the findings of this study are included within the article.

Disclosure

A preprint of this work has previously been published at Research Square [31].

Conflicts of Interest

The authors declare that they have no conflicts of interest.

Acknowledgments

This study was supported by Huawei Research Fund (no. HF2018055044).

References

- [1] N. P. Jouppi, C. Young, N. Patil et al., "In-datacenter performance analysis of a tensor processing unit," *ACM SIGARCH-Computer Architecture News*, vol. 45, no. 2, pp. 1–12, 2017.
- [2] T. Chen, Z. Du, N. Sun et al., "DianNao: a small-footprint high-throughput accelerator for ubiquitous machine-learning," *Architectural Support for Programming Languages and Operating Systems*, vol. 49, no. 4, pp. 269–284, 2014.
- [3] Y. Chen, T. Luo, S. Liu et al., "DaDianNao: a machine-learning supercomputer," in *Proceedings of the International Symposium on Microarchitecture*, pp. 609–622, Cambridge, UK, December 2014.
- [4] Z. Du, R. Fasthuber, T. Chen et al., "ShiDianNao: shifting vision processing closer to the sensor," *International Symposium on Computer Architecture*, vol. 43, no. 3, pp. 92–104, 2015.
- [5] D. Liu, T. Chen, S. Liu et al., "PuDianNao: a polyvalent machine learning accelerator," *ACM SIGARCH-Computer Architecture News*, vol. 43, no. 1, pp. 369–381, 2015.
- [6] S. Liu, Z. Du, J. Tao et al., "Cambricon: an instruction set architecture for neural networks," *ACM SIGARCH-Computer Architecture News*, vol. 44, no. 3, pp. 393–405, 2016.
- [7] R. Andri, L. Cavigelli, D. Rossi, and L. Benini, "YodaNN: an architecture for ultra-low power binary-weight CNN acceleration," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 99, p. 1, 2017.
- [8] B. Moons, R. Uytterhoeven, W. Dehaene, and M. Verhelst, "14.5 envision: a 0.26-to-10TOPS/W subword-parallel dynamic-voltage-accuracy-frequency-scalable convolutional neural network processor in 28nm fdsoi," in *Proceedings of the Solid-State Circuits Conference*, San Francisco, CA, USA, February 2017.
- [9] B. Moons and M. Verhelst, "A 0.3-2.6 TOPS/W precision-scalable processor for real-time large-scale ConvNets," in *Proceedings of the IEEE Symposium on VLSI Circuits*, Honolulu, HI, USA, June 2016.
- [10] S. Yin, P. Ouyang, J. Yang et al., "An ultra-high energy-efficient reconfigurable processor for deep neural networks with binary/ternary weights in 28NM CMOS," in *Proceedings of the 2018 IEEE Symposium on VLSI Circuits*, pp. 37–38, Honolulu, HI, USA, June 2018.
- [11] D. Kim, J. Kung, S. Chai, S. Yalamanchili, and S. Mukhopadhyay, "Neurocube: a programmable digital neuromorphic architecture with high-density 3D memory," in *Proceedings of the International Symposium on Computer Architecture*, Seoul, Republic of Korea, June 2016.
- [12] P. Chi, S. Li, C. Xu et al., "PRIME: a novel processing-in-memory architecture for neural network computation in ReRAM-based main memory," *ACM SIGARCH-Computer Architecture News*, vol. 44, no. 3, pp. 27–39, 2016.
- [13] S. Ali, A. Nag, N. Muralimanohar et al., "ISAAC: a convolutional neural network accelerator with in-situ analog arithmetic in crossbars," *International Symposium on Computer Architecture*, vol. 44, no. 3, pp. 14–26, 2016.
- [14] L. Song, X. Qian, H. Li, and Y. Chen, "PipeLayer: a pipelined ReRAM-based accelerator for deep learning," in *Proceedings of the IEEE International Symposium on High Performance Computer Architecture*, pp. 541–552, Austin, TX, USA, February 2017.
- [15] Y.-H. Chen, T. Krishna, J. S. Emer, and V. Sze, "Eyeriss: an energy-efficient reconfigurable accelerator for deep convolutional neural networks," *IEEE Journal of Solid-State Circuits*, vol. 52, no. 1, pp. 127–138, 2017.
- [16] S. Zhang, Z. Du, L. Zhang et al., "Cambricon-x: an accelerator for sparse neural networks," in *Proceedings of the International Symposium on Microarchitecture*, pp. 1–12, Taipei, Taiwan, October 2016.
- [17] S. Han, X. Liu, H. Mao et al., "EIE: efficient inference engine on compressed deep neural network," *ACM SIGARCH-Computer Architecture News*, vol. 44, no. 3, pp. 243–254, 2016.
- [18] A. Parashar, M. Rhu, A. Mukkara et al., "SCNN: an accelerator for compressed-sparse convolutional neural networks," *International Symposium on Computer Architecture*, vol. 45, no. 2, pp. 27–40, 2017.
- [19] J. Albericio, P. Judd, T. Hetherington, T. Aamodt, N. E. Jerger, and A. Moshovos, "Cnvlutin: ineffectual-neuron-free deep

- neural network computing,” *ACM SIGARCH-Computer Architecture News*, vol. 44, no. 3, pp. 1–13, 2016.
- [20] P. Judd, J. Albericio, T. H. Hetherington, M. A. Tor, and A. Moshovos, “Stripes: bit-serial deep neural network computing,” in *Proceedings of the International Symposium on Microarchitecture*, pp. 1–12, Taipei, Taiwan, October 2016.
 - [21] M. Gao, J. Pu, X. Yang, M. Horowitz, and C. Kozyrakis, “TETRIS: scalable and efficient neural network acceleration with 3d memory,” *ACM SIGARCH-Computer Architecture News*, vol. 45, no. 1, pp. 751–764, 2017.
 - [22] H. Li, X. Fan, L. Jiao, W. P. Cao, X. Zhou, and L. Wang, “A high performance FPGA-based accelerator for large-scale convolutional neural networks,” in *Proceedings of the International Conference on Field Programmable Logic and Applications (FPL)*, pp. 1–9, Lausanne, September 2016.
 - [23] J. Qiu, J. Wang, S. Yao et al., “Going deeper with embedded FPGA platform for convolutional neural network,” in *Proceedings of the International Symposium on Field-Programmable Gate Arrays*, pp. 26–35, New York, NY, USA, February 2016.
 - [24] H. Sharma, J. Park, D. Mahajan et al., “From high-level deep neural models to FPGAs,” in *Proceedings of the International Symposium on Microarchitecture*, pp. 1–12, Taipei, Taiwan, October 2016.
 - [25] R. Zhao, W. Song, W. Zhang et al., “Accelerating binarized convolutional neural networks with software-programmable FPGAs,” in *Proceedings of the International Symposium on Field-Programmable Gate Arrays*, pp. 15–24, New York, NY, USA, February 2017.
 - [26] B. Reagen, P. Whatmough, R. Adolf et al., “Minerva: enabling low-power, highly-accurate deep neural network accelerators,” *ACM SIGARCH-Computer Architecture News*, vol. 44, no. 3, pp. 267–278, 2016.
 - [27] T. Chen, T. Moreau, Z. Jiang et al., “TVM: an automated end-to-end optimizing compiler for deep learning. operating systems design and implementation,” in *Proceedings of the 13th USENIX Conference on Operating Systems Design and Implementation*, pp. 579–594, Berkeley, CA, USA, October 2018.
 - [28] T. Moreau, T. Chen, Z. Jiang, L. Ceze, C. Guestrin, and A. Krishnamurthy, “VTA: an open hardware-software stack for deep learning,” 2018, <https://arxiv.org/abs/1807.04188>.
 - [29] P. R. Panda, “SystemC-a modeling platform supporting multiple design abstractions,” in *Proceedings of the International Symposium on System Synthesis*, Montreal, QC, Canada, September 2001.
 - [30] X. Liang, *Ascend AI Processor Architecture and Programming*, Tsinghua University Press, Beijing, China, 2019.
 - [31] J. Hu, X. Zhang, and X. Shi, “A configurable simulator for neural network processors,” 2021, <https://www.researchsquare.com/article/rs-202912/v1>.