# SmartVM: A Smart Contract Virtual Machine for Fast On-Chain DNN Computations

Tao Li, Yaozheng Fang, Ye Lu, Jinni Yang, Zhaolong Jian, Zhiguo Wan, and Yusen Li

**Abstract**—Blockchain-based artificial intelligence (BC-AI) has been applied for protecting deep neural network (DNN) data from being tampered with, which is expected to further boost trusted distributed AI applications in many fields. However, due to smart contract execution environment architectural defects, it is challenging for previous BC-AI systems to support computing-intensive tasks on-chain performing such as DNN convolution operations. They have to offload computations and a large amount of data from blockchain to off-chain platforms to execute smart contracts as native code. This failure to take advantage of data locality has become one of the major critical performance bottlenecks in BC-AI system. To this end, in this article, we propose SmartVM with optimization methods to support on-chain DNN inference for BC-AI system. The key idea is to design and optimize the computing mechanism and storage structure of smart contract execution environment according to the characteristics of DNN such as high computational parallelism and large data volume. We decompose SmartVM into three components: 1) a compact DNN-oriented instruction set to describe computations in a short number of instructions to reduce interpretation time. 2) a memory management mechanism to make SmartVM memory dynamic free/allocated according to the size of DNN feature maps. 3) a block-based weight prefetching and parallel computing method to organize each layer's computing and weights prefetching in a pipelined manner. We perform the typical image classification in a private Ethereum blockchain testbed to evaluate SmartVM performance. Experimental results highlight that SmartVM can support DNN inference on-chain with roughly the same efficiency against the native code execution. Compared with the traditional off-chain computing, SmartVM can speed up the overall execution by **70**×, **16**×, **11**×, and **12**× over LeNet5, AlexNet, ResNet18, and MobileNet, respectively. The memory footprint can be reduced by **84%**, **90.8%**, **94.3%**, and **93.7%** over the above four models, while offering the same level model accuracy. This article sheds light on the design space of the smart contract virtual machine for DNN computation and is promising to further boost BC-AI applications.

**Index Terms**—Deep neural network, smart contract, virtual machine, architectural support technology

## 1 Introduction

Blockchain-based artificial intelligence (BC-AI) has been a new researching hotspot [1], [2], [3], expected to boost trusted distributed AI training and inference [4], [5], [6], such as protecting deep neural network (DNN) data from being tampered [7], [8]. Smart contract is a piece of code which can be deployed on blockchain for executing application logic [9], [10]. Various blockchains have provided execution environment or virtual machine, such as Ethereum Virtual Machine (EVM) [11], [12], for interpreting and executing smart contract. The execution on virtual machine of the smart contract deployed on the blockchain is called on-chain computing and conducting the smart contract out of the virtual machine is correspondingly called off-chain computing [13], [14], [15].

The existing main stream smart contract virtual machines have limited BC-AI application scope and further


- *Tao Li is with the College of Computer Science, Nankai University, Tianjin 300071, China, with the College of Cyber Science, Nankai University, Tianjin 300071, China, with the State Key Laboratory of Computer Architecture, Institute of Computing Technology, Chinese Academy of Sciences, Beijing 100045, China, and also with the Tianjin Key Laboratory of Network and Data Science Technology, Tianjin 300071, China. E-mail: litao@nankai.edu.cn.*
- *Yaozheng Fang and Zhaolong Jian are with the College of Computer Science, Nankai University, Tianjin 300071, China, and also with the Tianjin Key Laboratory of Network and Data Science Technology, Tianjin 300071, China. E-mail: {fyz, jianzhaolong}@mail.nankai.edu.cn.*
- *Ye Lu is with the College of Computer Science, Nankai University, Tianjin 300071, China, with the College of Cyber Science, Nankai University, Tianjin 300071, China, with the Information Security Evaluation Center of Civil Aviation, Civil Aviation University of China, Tianjin 300300, China, with the State Key Laboratory of Computer Architecture, Institute of Computing Technology, Chinese Academy of Sciences, Beijing 100045, China, and also with the Tianjin Key Laboratory of Network and Data Science Technology, Tianjin 300071, China. E-mail: luye@nankai.edu.cn.*
- *Jinni Yang is with the College of Cyber Science, Nankai University, Tianjin 300071, China, and also with the Tianjin Key Laboratory of Network and Data Science Technology, Tianjin 300071, China. E-mail: tol2020_nk@foxmail.com.*
- *Zhiguo Wan is with Zhejiang Lab, Hangzhou, Zhejiang 311121, China. E-mail: zhiguo_wan@163.com.*
- *Yusen Li is with the College of Computer Science, Nankai University, Tianjin 300071, China. E-mail: liyusen@nbjl.nankai.edu.cn.*


*Manuscript received 1 Dec. 2021; revised 19 May 2022; accepted 20 May 2022. Date of publication 0 . 2022; date of current version 0 . 2022.*
*This work was supported in part by CCF-AFSG Research Fund under Grant CCF-AFSG RF20210031, in part by the Special Funding for Excellent Enterprise Technology Correspondent of Tianjin under Grant 21YDTPJC00380, in part by National Natural Science Foundation under Grant 62002175, in part by the Open Project Fund of State Key Laboratory of Computer Architecture, Institute of Computing Technology, Chinese Academy of Sciences under Grant CARCHB202016, in part by the Key Research Project of Zhejiang Lab under Grant 2021KF0AB04, in part by the Natural Science Foundation of Tianjin under Grant 20JCZDJC00610, in part by the Open Project Foundation of Information Security Evaluation Center of Civil Aviation, Civil Aviation University of China under Grant ISECCA-202102, in part by the People's Republic of China ministry of education science and technology development center under Grant 2019J02019, and in part by Tianjin Graduate Scientific Research Innovation Project under Grant 2021YJSB014.*
*(Corresponding author: Ye Lu.)*
*Recommended for acceptance by J. Zhai.*
*Digital Object Identifier no. 10.1109/TPDS.2022.3177405*

development, since previous they cannot process complex tasks. For example, although the smart contract virtual machines such as EVM sustain more than 3,200 kinds of Dapps [15], there is no DNN application that can run on the blockchain [16]. DNN inference as yet cannot be directly and efficiently performed on blockchain by smart contract [17], [18]. The primary reason is that the smart contract execution environment in previous BC-AI system lacks operators, instructions and corresponding mechanism to support redundant complex DNN operations with high computational and memory complexity.

These issues lead to the existing BC-AI applications on blockchain that can only simply store a large amount of DNN weight data as a database. Computing-intensive tasks such as DNN convolutions have to be offloaded to the off-chain platform, executed as native code, and still need to download weight data from the blockchain. Unfortunately, downloading data is one of the most critical performance bottlenecks in traditional blockchain-based AI systems, which usually requires tens of thousands interface invoking and large latency.

In view of the above problems, on-chain computing turns out to be a convenient alternative and can lead to several benefits in terms of close to data source, avoiding data download latency and trusted execution, etc. Many previous related works in other areas have pointed out that, the better design for distributed systems is to move computation tasks to where the data is [19], [20]. Therefore, to take advantage of data locality, on-chain computing as the *move computation to data* paradigm is more natural for DNN inference.

Both academia and industry have paid attention to DNN inference on smart contract virtual machine [21]. They make explorations that allow sustaining the computational burden of DNN inference on the blockchain. The explorations aim at providing trusted computing processes, fueling intelligent applications without high latency, and conducting complex computations for BC-AI systems. For instance, Kim *et al.* [22] have processed DNN inference on-chain, but they utilize the mature JavaScript Virtual Machine rather than the most commonly used smart contract engine EVM for blockchain. Konstantin Kladko gives a hypothetical example in Ethresearch[1] and he describes a decentralized, trusted, fair and automatic Uber which runs a neural network based on driver history behaviors with smart contracts, to explain the advantages of running DNN on EVM. Nonetheless, these two examples have not been implemented in reality, since they both cannot meet the challenges that in order to perform convolution operations, DNN usually requires high computing power and an amount of memory space to store lots of immediate results.

In fact, introducing DNN computing to previous EVM on the blockchain can pose several difficult systemic challenges. First, there are no specific operation instructions, meanwhile, the general EVM instruction set will generate tens of millions of instructions for DNN inference. Interpreting and executing so many instructions will take a lot of time, which cannot satisfy the requirements of real applications. Second, all the operations on EVM are executed serially and the serial execution will also bring higher latency [23], [24], [25]. A large number of convolution computations and weight fetching from EVM storage in serial are so time-consuming without a parallel computing mechanism. For example, a single image inference over LeNet-5 on EVM can take more than 2.5 seconds, while the most common DNN applications only need dozens of milliseconds [26]. Third, the existing virtual machine EVM architecture designed for running small-scale programs has no runtime memory space management mechanism during smart contract execution [23]. DNN (e.g., AlexNet, ResNet) inference cannot run on the EVM solidly without memory overflow, because the inference will cause a high memory footprint, and lead to the Out-of-Memory exception in common resource-limited devices. To meet these challenges, fast on-chain DNN computation requires fine-grained architecture level design and corresponding mechanism support.

To this end, we present SmartVM, a new smart contract virtual machine for fast on-chain DNN computations. SmartVM can also enable smart contract execution on heterogeneous devices such as GPU, and offer roughly the same executing performance compared with CPU/GPU. The key idea is to provide specific instructions and multiple optimization mechanisms and techniques for the complex inference process of DNN. Our novel contributions in this paper can be summarized as follows:

- We design DNN-oriented domain-specific instructions having a strong descriptive capability for DNN. Compared with running under EVM, the DNN inference efficiency in SmartVM with the proposed instructions can be accelerated by up to **38**×.
- We propose a dynamic memory management method by designing the Buffer technique on EVM memory at runtime. The proposed mechanism realizes the physical RAM space multiplexing, since the Buffer can adjust size flexibly to store only one layer's feature maps rather than all layers' feature maps. This mechanism can significantly reduce the RAM footprint by **90.7%** on average.
- We propose the block-based weight prefetching method and parallel computing mechanism. The weight data can be prefetched and loaded in block-wised rather than a single value, and the times of memory access can also be reduced. These approaches can hide the execution waiting time and improve computing efficiency by **13.1%** on average.
- We implement SmartVM by embedding it as a blocking component into Ethereum as a smart contract virtual machine, and we evaluate SmartVM by conducting typical image classification tasks in a real private Ethereum platform. Compared with DNN inference on CPU, the experimental results highlight that SmartVM can support DNN inference on-chain with roughly the same efficiency against the native code execution.

## 2 Background and Motivation

In this section, we draw our motivations and key idea about SmartVM design from two aspects. First, we give some

1. https://ethresear.ch/

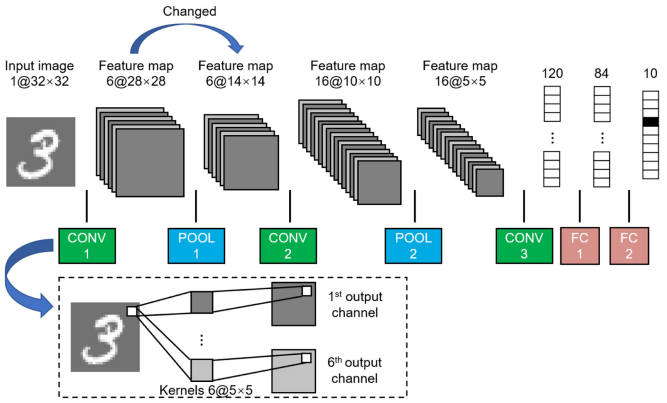



Fig. 1. LeNet-5 architecture of inference.

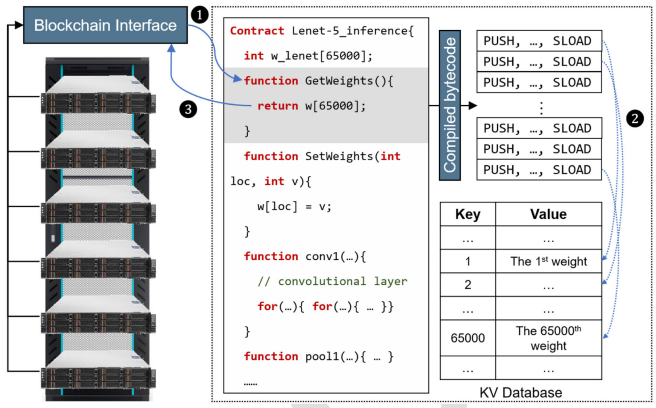



Fig. 2. CNN Computing Process in Typical BC-AI.

preliminary concepts of DNN (in the particular, convolutional neural network, CNN) and its main characteristics. In this paper, we use one of the representative DNN, CNN, to show the characteristics of DNN, because CNN is one of the most widely used DNNs, and the previous works also use CNN to represent DNN such as [27]. Second, we give the scenarios of on-chain CNN computing to show the motivation of SmartVM from the application respect. Third, we analyze the CNN inference process performance under the traditional typical BC-AI architecture to point out the disadvantages and shortcomings of off-chain computing. Lastly, we elaborate on the existing limitations and challenges of the complex computing on Ethereum Virtual Machine (EVM). In order to explain the details, we take LeNet-5 [28] as an example to conduct a breaking down analysis about the performance of on-chain CNN inference.

## 2.1 Convolutional Neural Network

Convolutional neural network (CNN) is a kind of DNN, which is widely applied in image recognition and classification [29], [30], [31]. As shown in Fig. 1, CNN architecture contains three types of layers: convolutional layer, pooling layer, and fully-connected layer. The input and output of each layer are called feature maps [32], [33]. The simple introduction of each kind of layer is as follows:

- The convolutional layer uses some weights (convolutional kernel) to perform enormous repetitive convolution operations to its input feature map. This layer extracts the high-level features of the input feature map. Convolution operations account for more than 90% of CNN computations which are also massive.
- The pooling layer usually appears after the convolutional layer. The pooling layer is responsible for reducing the size of input feature map to decrease the computational power required to process the image.
- The fully-connected layer multiplies its input feature map by a weight matrix and then adds a bias vector. This layer is used for learning non-linear combinations of the high-level features as represented by the output of the convolutional layer.

It is widely known that a well-trained CNN usually has a large number of weights [34], for example, VGG-16 has 130 million weights [35]. Using such CNNs to conduct image recognition task requires high memory space (16GB). The CNNs are getting explosively deeper (i.e., more layers) and wider (i.e., more parameters per layer) for higher modeling capacities. The number of weights can be increased rapidly such as the ViT network [36] with about 2,000 million weights presented in October 2020.

From the above preliminary explanation, we can obtain three main observations about CNN computing. Firstly, the size of feature maps is different before and after a layer's computing. In LeNet-5, the size of the feature map is increased after Conv1, Conv2, and FC1, while decreased after Pooling1, Pooling2, and FC2. Secondly, there are many identical convolutions with each other in each convolution layer in CNN. Thirdly, for a specific layer, the convolution operations and weight fetching are data-independent, thus fetching the next output channel's weight can be executed in advance when computing the current layer of CNN.

## 2.2 Problems of Off-Chain Computing

We give the typical BC-AI architecture in Fig. 2 to ease understanding the problems of CNN computing off-chain. The BC-AI architecture adopts on-chain weight storage and off-chain inference.

In typical blockchains such as Ethereum, each smart contract maintains a storage trie to record the CNN weights. Each weight is stored as a leaf node in storage trie. Each leaf node is stored in a key-value database as a single item [23]. Each variable is a leaf node in the storage trie, so the weights will be recorded to the storage trie as leaf nodes. The storage trie is a data structure logically living in RAM. When a smart contract is invoked, the storage trie will be loaded from the key-value database (in hard disk) to RAM.

Although there are some researches ongoing to deploy optimizations on hardware accelerator to execute smart contract off-chain, the approaches cannot match the bytecode execution mode and need rewriting a large number of smart contracts by native code [17]. In addition, the CNN weights have to be downloaded from the database in RAM as usual by invoking the corresponding smart contract [37]. As shown in Fig. 2, weight fetching will invoke the `get()` function (❶) and the function is compiled down to more than 50 instructions, including `SLOAD` and other instructions for reading the weights from the database (❷). Then the weights are returned to the off-chain platforms (❸). Note that the weights have to be fetched by EVM instructions with a single

thread one by one because they are stored discontinuously and independently with each other in database. Therefore, this small-grained way of fetching weights creates a bottleneck, and it is more than thousands of times that a smart contract with so many EVM instructions invokes `get()` function and accesses the database to fetch weights. Such invocations and access can cause high latency and make the data fetching incredibly time-consuming.

We have deployed experiments before to observe fetching DNN data from the Ethereum blockchain to off-chain platforms. For instance, regarding the LeNet-5, fetching weight needs about 65,000 times invocation of database interface, the latency is more than 3,000ms. For the AlexNet, the same process invokes data reading interfaces about 62,000,000 times and the latency is up to 3,480,000ms. In addition, CNN weight should be often updated along with the changes in AI applications in practice, so the weight downloading which takes so long time is frequent and inevitable. And even worse, all the data is computed outside the trusted computing environment, which is insecure and vulnerable to be tampered with [38], and also deviates from BC-AI original design intention about trusted computing.

Consequently, the time-consuming weight downloading is one of the most critical performance bottlenecks in BC-AI systems. With the fast increase of CNN model size, the weight downloading time is gradually longer and longer. Due to the single-thread design of EVM, the operations for fetching weights are executed serially which also has a serious negative effect on EVM performance.

### 2.3 Scenarios of On-Chain CNN Computing

The typical BC-AI systems are applied in many applications such as healthcare, model exchange, and smart transportation. But the previous works apply blockchain as a database to store data. All the typical applications in BC-AI can be supported by on-chain CNN computing. The typical applications in BC-AI includes but are not limited the digital asset evaluation, distributed AI model trade, and distributed computation above privacy data.

Firstly, on-chain NN computing can support secure distributed computing to enable blockchain-based AI model trade like Algorithmia DanKu.[2] The smart contract can be used for storing, executing, and validating the AI models. The smart contract-based model trade is more reliable and secure. Secondly, the on-chain NN computing can achieve trusted distributed computation above medical privacy data, the computation is performed in smart contract and achieves consensus of results among multi parts [39]. Besides, the traditional application such as UBER can also be deployed in on-chain computing environment. The drivers upload the driving data to blockchain, the smart contract can pay the drivers according to the drivers' behaviors based on trained model.

Previous work also considers that encoding the trained neural network inside a zkSNARK circuit to protect data security and computation security.[3] However, such method requires complex circuit experiences and preliminaries. The on-chain NN computing can be implemented by human-friendly programming languages and achieves the same effect as the circuit-based method.

Though the typical BC-AI applications are trusted and secure, the execution engine and environment of smart contract are low-performance, which limits more applications deployed on-chain. The proposed SmartVM provides a smart contract virtual machine to support on-chain CNN computing in high performance.

The CNN computation includes training and inference. This paper focuses on the CNN inference, because in the traditional BC-AI systems, the trained model needs to be deployed in blockchain. However, the on-chain inference based on the deployed model is low-performance. Therefore, from the application respect, we focus on CNN inference, as the existing blockchain and smart contract architecture can not support inference in high-performance. The model training includes forward propagation and back propagation, the performance bottleneck of training mainly appeared in back propagation. Specifically, the gradient and temporary data communication and storage bring high latency. The training is usually performed on high-performance platforms (e.g., cloud server), so the training is performed offline and off-chain based on local data. Blockchain is often used as a database and data source, when the blockchain is applied in model training, the blockchain can protect data from tampering, but can not improve training accuracy. Besides, the training mainly focuses on the network architecture, Big Data movement, and model accuracy. But the SmartVM is designed to support high-performance on-chain CNN computing through architectural design. In the future, the SmartVM can support training through communication optimization for Big Data. Therefore, we focus on the inference, and the training is out of the scope of our work.

### 2.4 Limitations and Challenges of On-Chain Computing

The traditional process of on-chain inference in detail summarized is shown in Fig. 3. The LeNet-5 neural network model is programmed by high-level contract-oriented language (e.g., Solidity), which can be compiled down to bytecode. The bytecode is executed in EVM interpreter, and the temporary data is stored in EVM Stack and EVM Memory. The LeNet-5 weights are stored in key-value database as described in Section 2.2. Before bytecode execution, the weights are organized as a trie and loaded into EVM Storage and each weight is stored as a node leaf of this trie. Storage is a specific block of physical RAM. When LeNet-5 inference, the Storage needs to be also accessed more than thousands of times. We pick EVM as our on-chain computing baseline, because the EVM is the most widely used contract execution environment. The original EVM is designed for simple financial functions and normal operations, which is not fit the CNN inference. Based on these preliminaries, we explain the limitations of on-chain CNN inference and central challenges of SmartVM design as follows:

First, since the EVM is Turing-complete, which means its smart contracts can solve any type of problem and perform any logical step of a computational function at least hypothetically [40], we consider running CNN on-chain can be realized both theoretically and technologically. However,

2. https://github.com/algorithmiaio/danku
3. https://github.com/ethereum/research/issues/3

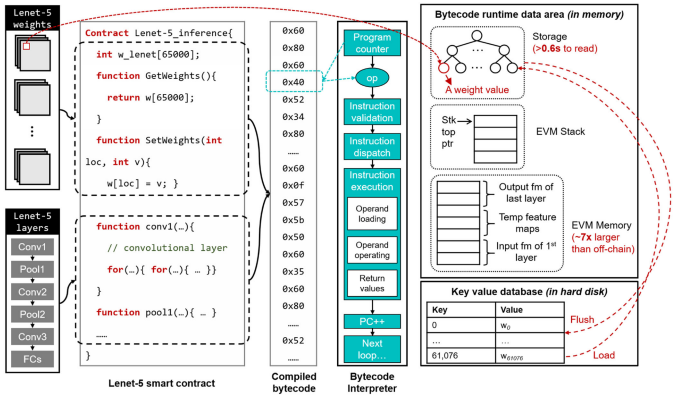



Fig. 3. On-chain LeNet-5 inference process.

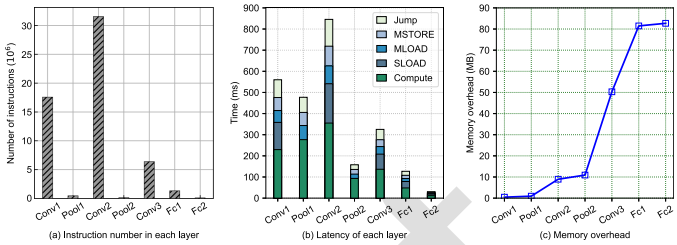



Fig. 4. The performance of LeNet-5 on-chain inference.

existing EVM architecture has no instructions to support highly efficient CNN inference. EVM instruction set is designed for general computing operations rather than the complex CNN computing operations, which cannot be described by a short number of instructions. As a result, as aforementioned, tens of millions of redundant instructions are generated and will affect smart contract performance according to our profiling. For instance, as shown in Figs. 4a and 4b, more instructions bring higher latency in the same kind of layer. And nearly 40% time in each LeNet-5 layer is used for fetching weights (`SLOAD` and `MLOAD`). Besides, the large number of instructions are limited by the gas mechanism of Ethereum [41]. Moreover, although the interpretation mechanism of EVM can be optimized indirectly such as EVMONE,[4] by precomputing the gas cost and stack requirements of the instructions, the performance improvements for complex computations are not sufficient and enough. Therefore, the first challenge of on-chain inference is to encounter the contradiction between CNN-oriented operator instruction lacking and general original instruction explosion in the previous smart contract execution virtual machine.

Second, EVM lacks memory management mechanism for processing the massive input data and immediate results during CNN computing. Specifically, the useless data in EVM memory is never freed, which causes high memory footprint and can not perform DNN solidly. For example, EVM always places new objects at the free EVM Memory pointer and these occupations will always be resident in the memory not be released .[5] In practice, the traditional memory management strategies like rolling array cannot satisfy the requirements of on-chain CNN inference. The traditional strategies are high-level solutions, and the compiled results are static. However, the memory required during CNN inference is variable and dynamic. Moreover, the memory management needs not only space compression or multiplexing, but also needs space scheduling, address conversion, and so on. The complex functions can not be realized by high-level solutions such as rolling array.

Although we can use `MSTORE` to malloc new EVM Memory space, it may cause unexpected errors (e.g., EVM Memory overlap) [42]. Even the LeNet-5 on-chain inference needs up to 90MB RAM (see Fig. 4c), but EVM memory supporting common smart contracts (e.g., ERC20) to perform is usually about one megabyte. The larger-scale neural networks can directly cause memory overflow exceptions. Furthermore, the number of CNN weight has been increased from 60,000 to 2,000,000,000 over the last 10 years. Running CNN requires more and more memory. Consequently, on-chain inference comes at a heavy memory burden challenge.

Third, in traditional EVM execution mechanism design such as single-thread, all the operations on EVM execute in serial mode. This implies that the CNN operation has to wait for the end of the weight fetching before it can be calculated. In fact, the structural feature of CNN is actually provided with high parallelism, the serial execution mode will obviously slow down inference performance. Moreover, the existing EVM lacks heterogeneous accelerating platforms such as GPUs supporting technology. It is worth noting that CNN inference requires yet data loading from EVM Storage or key-value database to EVM Stack, which needs thousands of times of accessing physical RAM memory to read data. Therefore, these time-consuming serial processes and technical defects strangle CNN computing on-chain. In summary, fast on-chain CNN computing is in desperate need of fine-grained architecture level design and corresponding mechanism support.

## 3 Basic Design of SmartVM

The key to applying SmartVM to achieve fast on-chain DNN computations is to efficiently interpret and execute the smart contract utilized for DNN computing. As mentioned before, on-chain CNN computing is challenging, considering the systemic limitations of operator instruction, memory footprint, and execution mechanism. Therefore, in this section, we propose SmartVM, to our best knowledge, the first architectural support technology aiming at speeding up on-chain CNN inference. Here, we first elaborate on the architecture overview of SmartVM. Then, we design the novel CNN-oriented specific instruction set for performing CNN operations in SmartVM. Next, we propose the dynamic memory space mechanism to reduce memory footprint during smart contract runtime. In the end, we present the optimization mechanism of block-based weight prefetching and computing towards making better use of smart contract parallel executing potentials, in order to further improve on-chain computing efficiency.

### 3.1 Architecture Overview

The overview of SmartVM is shown in Fig. 5. The SmartVM consists of the core, hardware interface, and data segment. Besides, SmartVM provides an extended CNN-oriented instruction set. The core is used for interpreting and

4. https://github.com/ethereum/evmone
5. https://docs.soliditylang.org/en/latest/

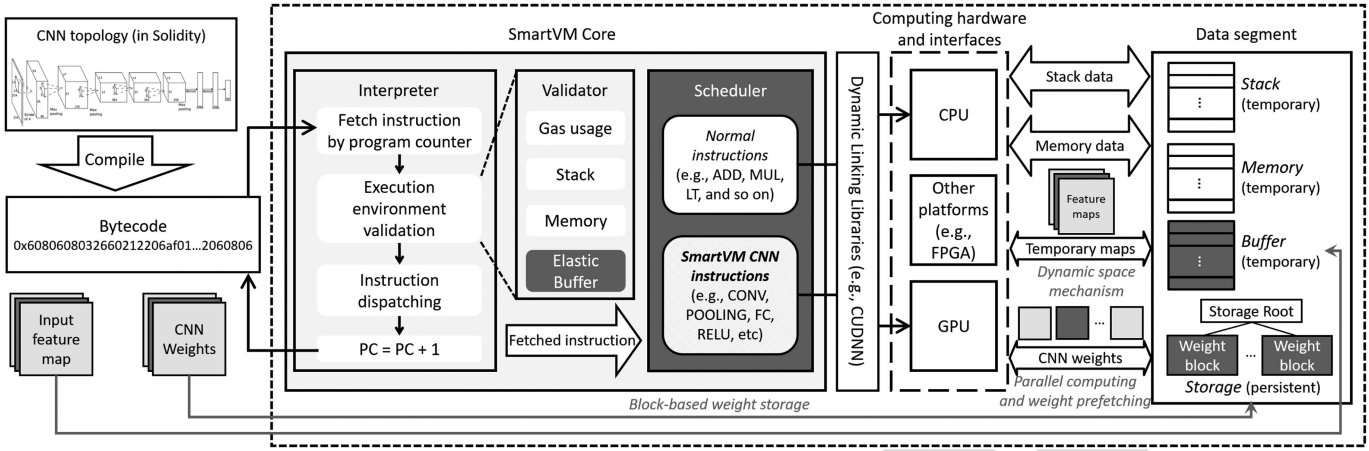



Fig. 5. SmartVM architecture overview.

dispatching instruction during CNN inference. The runtime data of inference is stored in the data segment.

A typical CNN network can be divided into two parts: CNN architecture and CNN weights (e.g., convolution kernel). The CNN architecture describes the number and the order of each kind of layer. The CNN architecture is implemented by a high-level smart contract, which is usually programmed by Turing-complete languages (e.g., Solidity). In SmartVM, the high-level smart contract will be compiled down to bytecode before inference. There are two types of instructions in the bytecode: the proposed CNN-oriented instruction and basic instruction (e.g., ADD, MUL, etc.). The CNN weights are stored in the blockchain's persistent key-value database. At contract runtime, the CNN weights will be loaded into memory as a cache. In SmartVM, weight fetching time can be reduced by decreasing the times of reading cache by the proposed block-based weight storage method.

The core of SmartVM has three parts: bytecode interpreter, instruction validator, and computing platform dispatcher. The bytecode interpreter fetches the instruction from a given bytecode by the program counter. Before execution, the instruction validator checks the execution context (e.g., stack overflow). Once the context satisfies the condition of the instruction execution, the interpreter dispatches the instruction. The dispatching refers to jumping to the corresponding native code segment that implements the instruction.

The computing platform dispatcher can assign different kinds of instructions to different hardware to enable heterogeneous computation: by default, in SmartVM, the CNN-oriented instructions are assigned to GPU, while other instructions are assigned to CPU. Furthermore, other hardware (like FPGA) can be also supported through the shared libraries. In SmartVM, we have implemented three types of hardware interfaces to support the dispatcher transmitting instructions to the target platform.

Temporary data during contract execution is stored in the data segment. The Stack stores instruction operands, the Memory stores complex type data (e.g., array), and the Storage is used for storing CNN weights. According to the characteristics of CNN computing (see Section 2.1), in SmartVM, we design a dynamic memory management method, which provides space multiplexing for feature maps during CNN inference, to reduce the memory footprint by defining an elastic Buffer space in SmartVM Memory.

In this section, the CNN is an example to show the design of SmartVM. The SmartVM can also be extended to support other kinds of DNN such as recurrent neural network (RNN), which only needs to implement corresponding instructions and operations. The SmartVM is designed as a common architecture with general optimization methods. The proposed instruction set can be extended through configuration interfaces, and the storage scheme can also be costumed according to the characteristics of DNN.

### 3.2 CNN-Oriented Instructions

In SmartVM, we divide CNN-oriented instructions into two types: computational and data transfer instruction. CNN computational operations should be described succinctly and efficiently. The computational instruction encapsulates and fuses common CNN operators. The data transfer instructions support moving data from/to an area (such as SmartVM Buffer, EVM Memory) to/from another area.

Computational instruction can describe mainly three granular computation operations in CNN inference: a whole CNN architecture, a specific layer in CNN (e.g., convolutional layer, pooling layer), and atomic operations (e.g., matrix multiplication). It is obvious that the finer the granularity of a computational instruction, the better description capability it is. In SmartVM, in particular, packaging a layer's computation into one instruction can achieve relatively high computational efficiency. Besides, the common usage operators in AI frameworks such as BatchMatmul, Broadcast, and Transpose are all can be extended to the proposed instruction set through pre-defined interfaces.

The data transfer instructions are designed to support data moving operations about the Buffer. When invoking a smart contract for CNN inference, the input feature map data is stored initially in Memory, then the data can be moved from Memory to Buffer through the instructions. Once the whole CNN network inference is completed, the final output feature map data which is stored in Buffer should be moved to Stack as the return value of the invocation.

As shown in Table 1, we list some representative CNN-oriented instructions. Each instruction has a mnemonic and a unique hexadecimal opcode. Each instruction's function is given in the description column. For example, `Conv_TPD`

TABLE 1
CNN-Oriented Instructions in SmartVM

| Type | Name | Opcode | Description | Stack required (Key arguments) |
|---|---|---|---|---|
| Computation | CONV_SING | 0x21 | Implement single channel convolution | 8 (Kernel, Output channel, Stride) |
| (Convolution) | CONV_MUL | 0x22 | Implement multi-channel convolution | 8 (Kernel, Output channel, Stride) |
| | CONV_3D | 0x23 | Implement 3D convolution | 8 (Kernel, Output channel, Stride) |
| | CONV_TPD | 0x24 | Implement transposed convolution | 8 (Kernel, Output channel, Stride) |
| (Pooling) | POOL_MAX | 0x25 | Implement max pooling | 5 (Stride, Input channel) |
| | POOL_AVG | 0x26 | Implement average pooling | 5 (Stride, Input channel) |
| | POOL_OL | 0x27 | Implement overlapping pooling | 5 (Stride, Input channel) |
| (Full connected) | FULL_CON | 0x28 | Implement full connected layer | 5 (Input channel, Output channel) |
| | MAT_MUL | 0x29 | Implement matmul | 2 (Addresses of two matrix) |
| (Active) | ACT_SM0 | 0x2a | Implement softmax function | 1 (Value) |
| | ACT_SM1 | 0x2b | Implement Sigmoid function | 1 (Value) |
| | ACT_RL | 0x2c | Implement ReLU function | 1 (Value) |
| | ACT_TANH | 0x2d | Implement Tanh function | 1 (Value) |
| (Buffer) | BUF_SCL0 | 0x2e | Increase Buffer's data with specific times | 1 (Specific times) |
| | BUF_SCL1 | 0x2f | Reduce Buffer's data with specific times | 1 (Specific times) |
| | BUF_BIAS | 0x30 | Add Buffer's data and bias | 1 (Base address of bias) |
| Data transfer | MTOB | 0x31 | Transfer data from Memory to Buffer | 2 (Data offset) |
| | BTOM | 0x32 | Transfer data from Buffer to Memory | 2 (Data offset) |
| | BTOS0 | 0x33 | Transfer data from Buffer to Stack | 2 (Data offset, Size) |
| | BTOS1 | 0x34 | Transfer data from Buffer to Storage | 2 (Data offset, Size) |
| (Buffer set) | BUF_CLS | 0x35 | Clean Buffer's data | 1 (Clean number) |
| | BUF_FIL | 0x36 | Fill Buffer's data with specific data | 1 (Specific filled data) |
| | BUF_INIT | 0x37 | Initial Buffer with specific size | 1 (Specific size) |
| | BUF_ALLO | 0x38 | Allocate specific size to Buffer | 1 (Specific size) |
| | BUF_FREE | 0x39 | Free specific size from Buffer | 1 (Specific size) |
| | BUF_COPY | 0x3a | Copy a same Buffer | 2 (Start and end pointers) |

instruction is used for transposed convolutional computation. The stack required column defines the number of stack items that the instruction requires. For example, `Conv` requires eight items to store the arguments of convolutional computation, such as input channel number. During instruction execution, once the reminder stack space is less than required, the on-chain CNN inference will be interrupted and an exception will be thrown.

Note that the SmartVM supports all the operations in smart contract, including CNN-related and non CNN-related. In SmartVM, the CNN inference can be realized by only normal instructions, only CNN-oriented instructions, or the both. The CNN-oriented instructions are compiled to high-performance bytecode, the SmartVM will not compulsorily change the developers' preference, which also means that the SmartVM will not bother developers. The proposed CNN-oriented instructions coexist with native EVM instruction set in SmartVM compiler. The developers can program CNN programs both with and without CNN-oriented instructions. All the operations in user's smart contract can be recognized by SmartVM's compiler. When compiling the smart contract, the CNN-related operations in SmartVM are compiled down to high-performance bytecode. The high-performance bytecode includes the CNN-oriented instructions. And the normal operations are compiled down to normal bytecode (non high-performance). In conclusion, the SmartVM can support any kind of operators.

As shown in Fig. 6, we give an example to show the usage and workflow of proposed CNN-oriented instructions. A LeNet-5 architecture can be programmed by CNN-oriented instructions through the in-line assembly programming method in a high-level based smart contract. Then the smart contract is compiled down to bytecode. For example, `Conv(...)` represents the first convolutional layer, and it is compiled down to eight `PUSH` operations for pushing arguments to Stack, and one `Conv` operation for

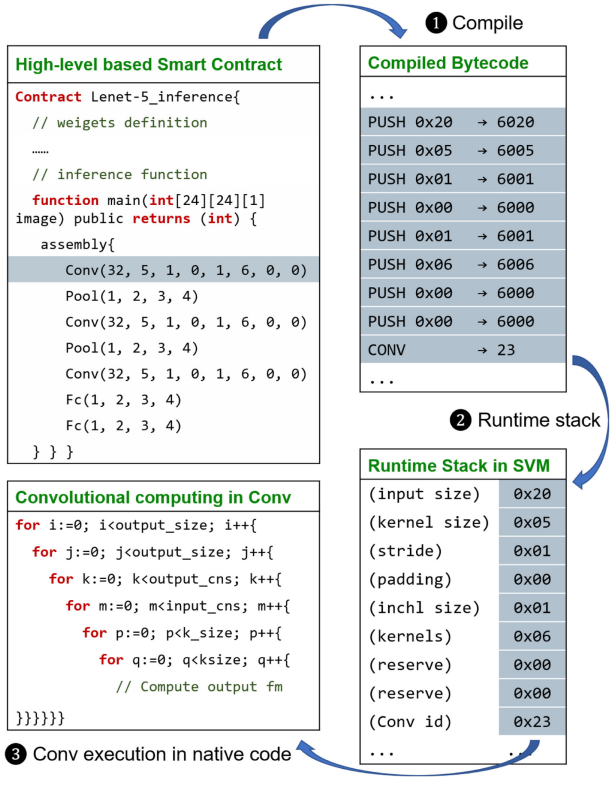



Fig. 6. The workflow of CNN-oriented instructions.

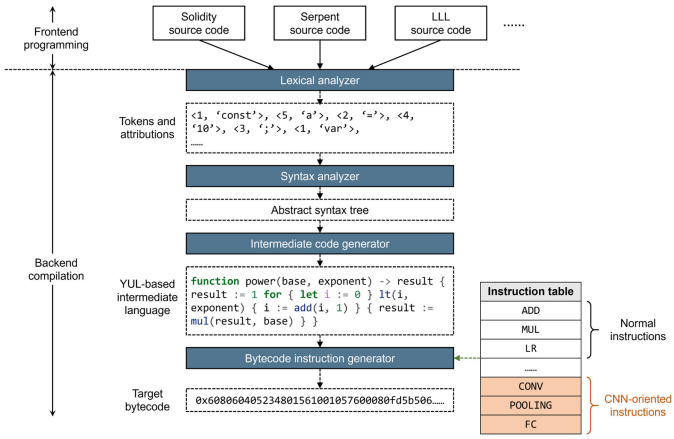



Fig. 7. The SmartVM compiler.

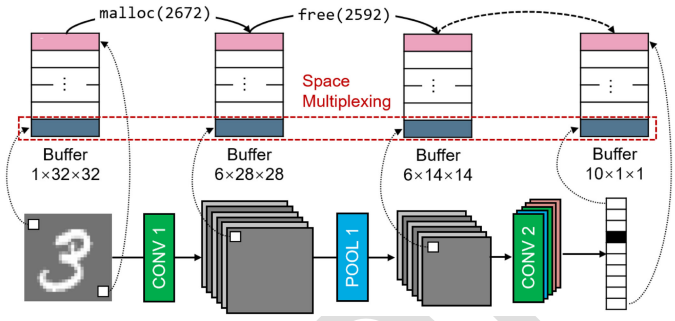



Fig. 8. The dynamic memory management method.

convolutional computation. In smart contract runtime, the parameters of the first convolutional layer are pushed onto Stack. In the native code wise, `Conv` instruction execution can be divided into five steps: 1) pop the arguments from Stack 2) fetch input feature map data from Buffer 3) fetch weights from Storage 4) perform convolutional computation 5) write back the output feature map back to Buffer.

In SmartVM design, from high-level smart contract to bytecode, a SmartVM compiler is provided for generating the CNN-oriented instructions. The compiler keeps an instruction table, which is the same as instruction set of smart contract virtual machine. Once the instruction set changes, the table should also be updated. As shown in Fig. 7, the compiler compiles high-level smart contract according to the instruction set. The source code is first parsed to abstract syntax tree (AST) by lexical and syntax analysis. Secondly, the AST is converted to Yul-based program (Yul is an intermediate language that can be compiled to bytecode for different backends). Then the Yul-based program is compiled to low-level bytecode according to the instruction table. For example, the code `sstore(v, zero)` in Yul is compiled down to `PUSH v, PUSH zero, SSTORE`. In the above steps, the exception handle part handles the exceptions during compilation. In SmartVM, the table in SmartVM compiler includes CNN-oriented instructions, and the CNN operations in high-level language can be corresponding compiled down to high-performance bytecode, while the normal compiler (e.g., Solc compiler) will give poor-performance bytecode (because the table has no CNN-oriented instructions).

### 3.3 Dynamic Memory Management Method

As aforementioned, although the EVM Memory can be used for storing runtime data, it will never be released dynamically during the contract execution which can cause a high memory footprint. This implies that to store CNN input and output feature maps, all the data are RAM-resident during the on-chain inference. Therefore, in SmartVM, in order to reduce the memory footprint, we propose a memory management method to provide dynamic memory allocation and release function in accordance with the feature map size of each CNN layer. As mentioned in Sec 2.3, the high-level smart contract language provides no library functions to manage memory in an automatic or manual manner. To make up for the defect, we partition a block from the memory and define it named as Buffer to store feature maps for each layer dynamically, and each Buffer item is set to 256 bits by default.

As shown in Fig. 8, we still take LeNet-5 inference as an example to explain the corresponding design details in SmartVM. The input feature map is stored in the partitioned Buffer and its size is 1024 (1*32*32), so the total size of Buffer at the present equals 1024. After the first convolutional layer computations, the output feature map size becomes 3456 $(6 * 24 * 24)$. The Buffer in consequence should be changed to be bigger by memory allocation. Then, the output feature map size becomes 864 $(6 * 12 * 12)$ after the computation of the first pooling layer, so the Buffer should be smaller by space release. According to the size of the intermediate results, we should dynamically change the corresponding memory size to prevent exceptions caused by the continuous growth of memory. In SmartVM, during CNN inference, the Buffer size upper and lower limits are decided by the output feature map size. The proposed memory management has two main insights: first, SmartVM can manage memory automatically and is compatible with all CNNs, because the Buffer is elastic according to the size of feature maps. Secondly, for some of the traditional high-level languages (e.g., C, C++), the developers may manage memory manually, our automatic method eases the developers and bring no extra burden to the developers. The pre-allocation and remapping approach is implemented by `append()` function, `malloc()` function, and `free()` function. Some high-level solutions (e.g., rolling array) are not fit the CNN inference, because the compiled results are static for the memory space, which can not fit the dynamic space requirements in CNN inference.

### 3.4 CNN Weight Prefetching and Parallel Computation

In CNN network, convolution computing can account for about 90% of the total processing work [43], [44]. In order to calculate convolution, each convolution operation should fetch weight from RAM to multiply the feature map data. This process will produce a large number of weight fetching operations. In the previous smart contract execution environment, the fetching process is in serial mode and so time-consuming. Therefore, we propose a block-based weight prefetching method in SmartVM to obtain more data once time to reduce the fetching time for CNN convolution calculation. Furthermore, because convolution operations when CNN inferences are repetitive and data-independent with

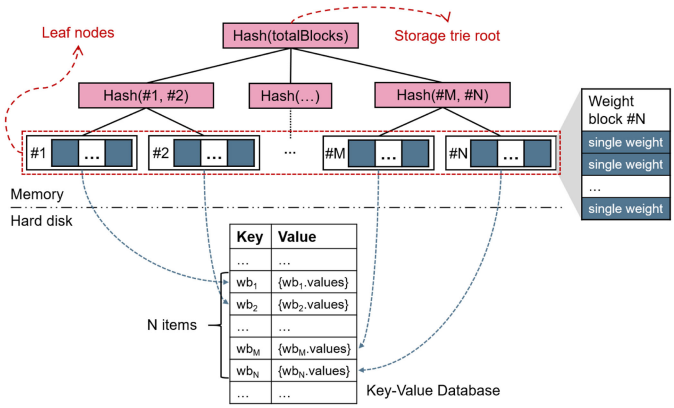



Fig. 9. The block-based weight storage.

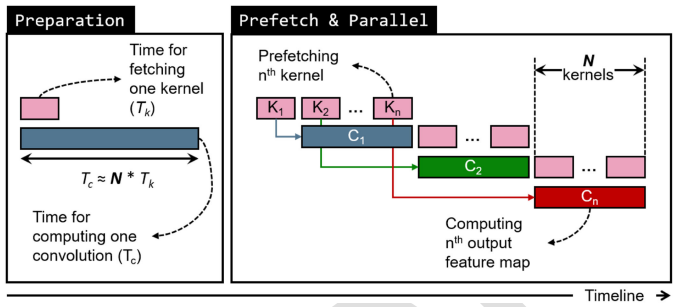



Fig. 10. The weights prefetching and parallel computation model in `Conv` instruction.

weight fetching at layer-wised, we also design a parallel computing mechanism to conduct weight fetching and convolution calculation at the same, in order to overcome the serial execution defect.

*Block-Based Weight Fetching*. Since reading data fast or slowly depends on the storage and organization of data to a certain extent, we first design how to store data reasonably. The block-based weight storage method is designed to reduce the number of reading data from physical memory, thereby speeding up the fetching process. To ease understanding about SmartVM design and previous blockchain storage structure, we take Ethereum as an example.

Before contract execution, the runtime EVM will load four tries from the persistent key-value database (e.g., Level-DB) on the hard disk into physical memory. The four tries are world state trie, receipt trie, transaction trie, and storage trie and are responsible for describing account information, transaction receipt, transaction information, and contract-related data, respectively. The storage trie stores the global variables of a smart contract, and each global variable is a leaf node of the trie. The CNN weights in a smart contract should be defined by global variables, so each weight data is a leaf node in the storage trie. As a result, when performing CNN inference within a smart contract, especially in convolutional and fully-connected layer's computation, it needs more than thousands of times to read the database to fetch weight data.

In SmartVM, we cluster and store weight data(e.g., a convolutional kernel) as a block rather than a single weight data in the leaf node of the storage trie (see Fig. 9). These weight blocks are indexed by a unique identifier in the corresponding smart contract data table in the database. Note that the number and the size of the weight block are not fixed and can be changed on demand. In a convolutional layer, a weight block may represent a convolutional kernel, while in the fully-connected layer it may represent a fully-connected matrix. For convolution computing which needs the whole convolutional kernel, SmartVM can fetch the convolutional kernel completely in the form of a block, and reduce the number of data reading thousands of times.

*Parallel Computation*. In the previous subsection, we explain how we design a block-based storage approach and enable it to process weight data fetching. Here, we further extend our design to explore more parallelism.

In the existing contract runtime, the serial mode during contract execution has limited complex computing potential, and will seriously degrade on-chain CNN inference performance. We have observed that the computing time in CNN inference is longer than the weight fetching time, and these two process tasks are data-independent in fact. We thus can perform fetching weight and CNN inference in the same instant. We reorganize the weights fetching process and the convolution computation of each channel in each layer in the pipeline manner. In a convolutional layer, the number of convolutional kernels equals the number of output feature maps. Concretely, we can prefetch the next output channel's convolutional kernel weight data when computing the CNN current output channel feature map.

As shown in Fig. 10, we define computing one single output channel's feature map time as $T_c$, and label fetching an output channel's convolutional kernel time as $T_k$. $T_c$ and $T_k$ are not constant in different convolutional layers and different CNNs, and normally, $T_k$ is less than $T_c$ in the convolutional layer. To initial parameters of $T_c$ and $T_k$, we record the time for fetching the first output channel's kernel and the time for the first output channel's convolutional computation. After that, we can calculate $\lfloor T_c/T_k \rfloor$ and define it as $N$, implying the number of output channels' convolutional kernel weight that can be prefetched maximally when computing convolution at the same time. When implementing SmartVM, we utilize two threads to conduct the two tasks in parallel. In this way, the weight fetching time can be covered by convolution computing time and thus further improving CNN inference performance on SmartVM.

## 4 Evaluation

To validate the design point of SmartVM and demonstrate its performance benefits, we have deployed experiments to build the BC-AI prototype system on the private Ethereum blockchain at CPU and GPU platform, which performing LeNet-5 over MNIST dataset, and AlexNet, ResNet18, and MobileNet over ImageNet [45] dataset, respectively. Besides, to prove the scalability of SmartVM, we perform experiments on the RNN (Recurrent Neural Network). The RNN used in this evaluation is LSTM (Long Short-Term Memory) with 28 cells and 55,296 weights. The objectives of the evaluation are fourfold: (1) testing the performance improvement of SmartVM compared with the offloading CNN weight data and computations as native code paradigm; (2) testing the performance improvement of SmartVM compared with traditional smart contract architecture regarding CNN inference; (3) providing

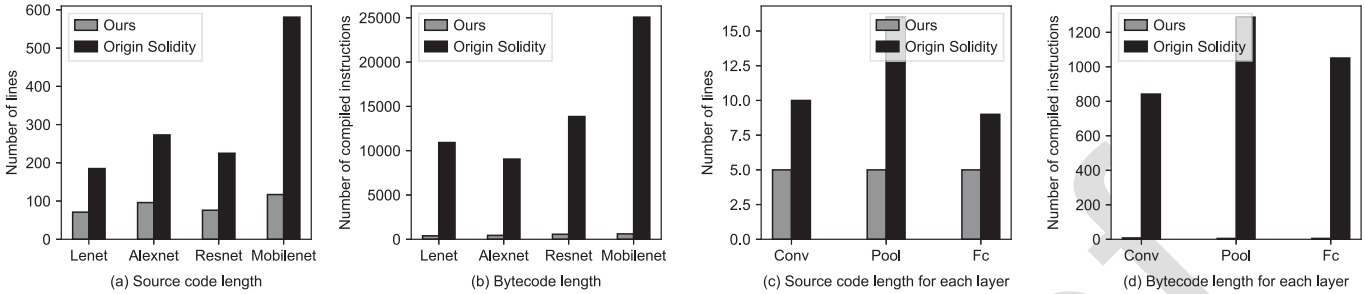



Fig. 11. The result of code length.

insights of SmartVM's outperforming its peers; and (4) studying the impact of SmartVM on the original BC-AI system.

### 4.1 Experimental Setup

*Hardware.* We deploy eight servers equipped with Xeon E5-2630 CPU (2.3GHz, 6 Cores) and 96GB memory to construct a private Ethereum network. The servers are connected with each other via a local area network by 1000Mb bandwidth. The GPU we utilized is NVIDIA GeForce 2080Ti.

*Metrics.* We compare SmartVM with traditional off-chain CNN inference in three aspects: inference latency, the RAM footprint, and the code length. Most CNN applications require low latency to achieve real-time inference with low computing resource overhead.

*Prototype.* Most of the existing work provides system model, but rarely provides source code (e.g., [22]). So we use the on-chain storage and off-chain computation model to represent existing works. The SmartVM baseline is off-chain model and Ethereum Virtual Machine. The private Ethereum network is implemented by Golang-based Ethereum (v1.7), and the smart contract execution environments are Ethereum Virtual Machine (v1.7) and SmartVM. The CNN smart contract is programmed by Solidity, and the corresponding compiler is based on Solc (v0.5.1). The CNN in native code is implemented by Golang (v1.14.2) and PyTorch (v1.10.0).

*Experimental Steps.* We deploy experiments for CNN inference on both off-chain and on-chain platforms. Performing off-chain CNN inference we need (1) Download CNN weights from Ethereum to the off-chain platform. (2) Performing CNN computation by CPU and GPU at local. Performing on-chain CNN inference we need (1) Fetching CNN weights from Storage. (2) Performing CNN computation by CPU and GPU in the corresponding smart contract (on-chain computing).

We give a fair comparison of performance between off-chain CNN inference and inference in SmartVM. We also analyze the reasons for the improvement of performance in detail.

### 4.2 Code Length

In the evaluation, the code length is picked as a metric for two reasons: the source code length is related to the convenience for development, and the bytecode length is related to the execution latency. Chen *et al.* point out that the code length is a meaningful metric only when the ISA is flexible enough to cover a broad range of applications in the target domain [46]. Note that in our evaluation for the code length, the comments are not included in our source file, and the lines of comments are not counted.

In this subsection, we focus on the comparison results between SmartVM instruction set and EVM instruction set in two aspects: the source code length (i.e., the number of Solidity smart contract source code) to show that the proposed CNN-oriented instructions can facilitate the programming, and the compiled bytecode to show that the CNN-oriented instructions can reduce the number of executed instructions in runtime in SmartVM. In the evaluation for code length, in order to keep fair, the language, compiler version, and other factors are kept the same. The experimental setup obeys the steps in [46]. Besides, we do not use any external library in our evaluation. The code length of source code refers to the line number (excluding blank lines) of source code file (counted manually). The code length of bytecode refers to the number of instructions (counted by the compiler automatically).

Fig. 11a shows that compared with the EVM instruction set, with the help of CNN-oriented instructions, programming MobileNet only needs 110 lines of source code in SmartVM, while this number is up to 600 in origin Solidity of EVM. Fig. 11c shows the number of source code lines for each kind of layer on average. Specifically, the SmartVM for programming convolutional layer, pooling layer, and FC layer is the same, while the EVM needs 2× to 3× to program the three layers because the computation logic is implemented only in a single instruction, and the instruction can be invoked by only one in-line assembly sentence (e.g., `assembly{conv (args)}`).

As shown in Fig. 11b, compared with the origin Solidity language which is EVM supported, SmartVM can reduce the compiled bytecode numbers by 95.8% on average with the proposed CNN-oriented instructions. Specifically, as shown in Fig. 11d, the number of bytecode instructions for the convolutional layer is eight, including `Conv` itself and seven `PUSH` for its parameters, while the number of original bytecode instructions is more than 840.

The off-chain Golang-based LSTM inference program is nearly 120 lines, and the on-chain Solidity-based smart contract needs nearly 110 lines. Fortunately, with the help of CNN-oriented instructions in SmartVM, the code length of the source code can be reduced to 27 lines. Furthermore, the compiled bytecode length can be reduced from nearly 4900 to 170, compared with native Solidity.

The reduced code length comes from the architectural design, the complex logic is achieved by the low-level instructions in low layer rather than the high-level program code. Though some high-level programming frameworks

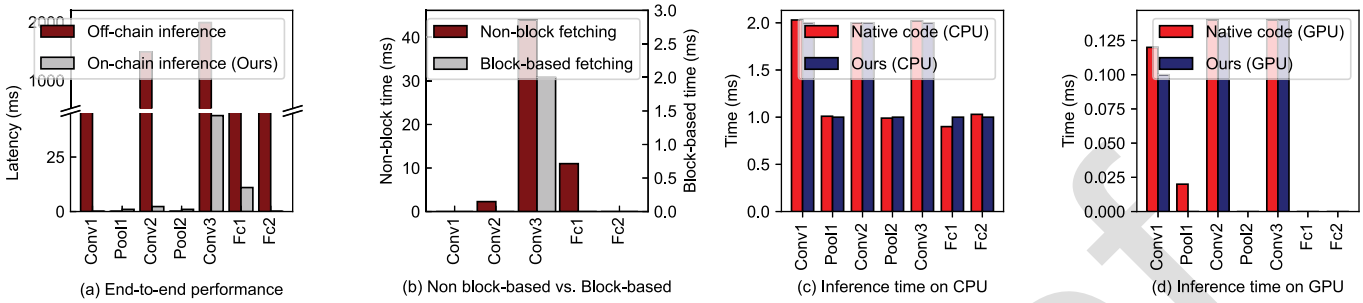



Fig. 12. Results for latency of LeNet-5 inference.

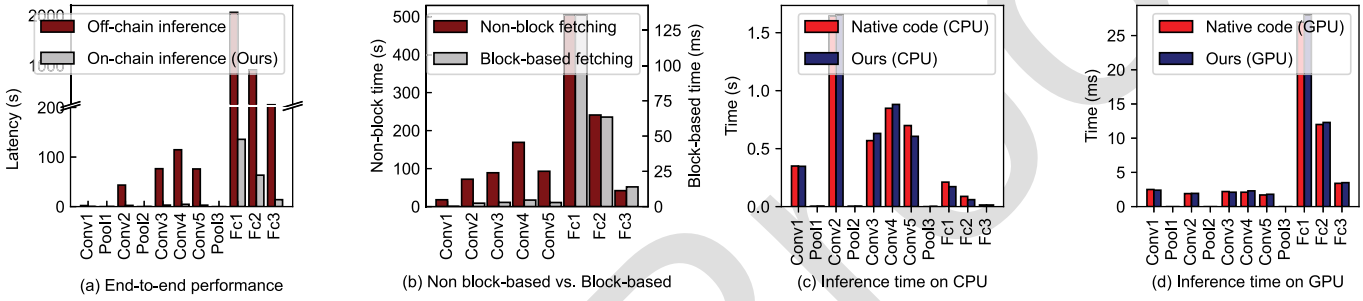



Fig. 13. Results for latency of AlexNet inference.

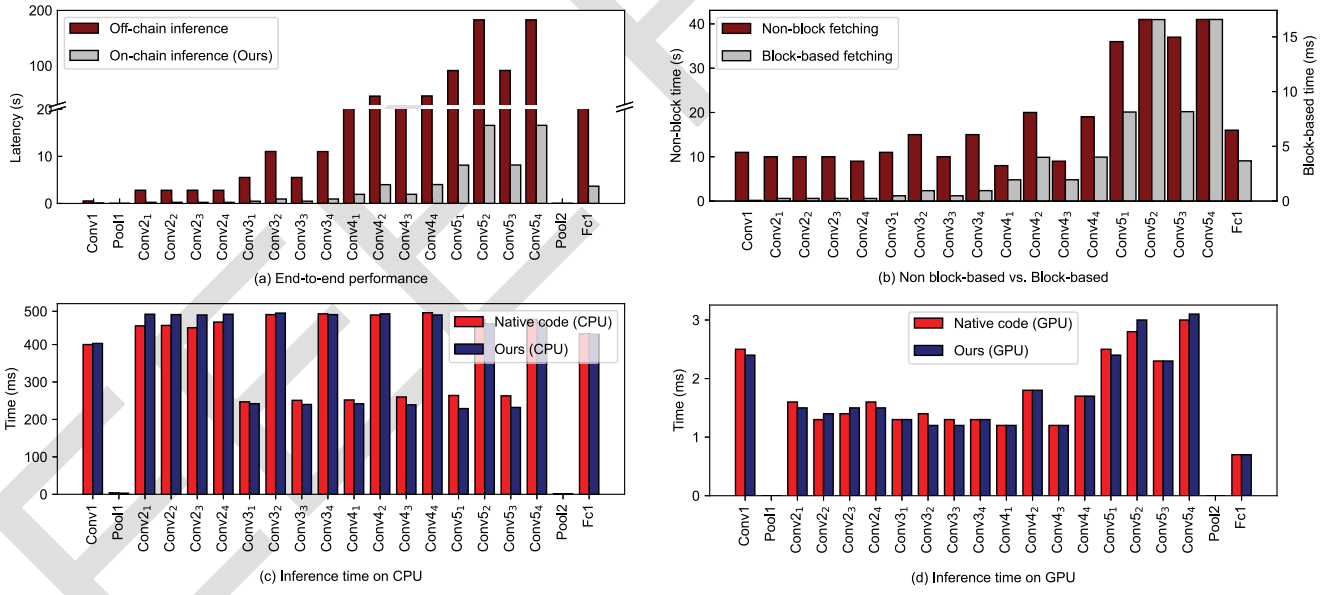



Fig. 14. Results for latency of ResNet-18 inference.

and libraries can also reduce the source code length, the compilation results are the same. The architectural and low-level instruction set design can support shorter code length with high-performance computation. Specifically, in fact, sometimes an implementation needs more lines of codes only due to the lack of abstraction and encapsulations. However, in SmartVM, the reduction of code length is mainly caused by the new low-level instructions rather than high-level language function library. A function library in high-level language cannot improve the execution performance, because the compiled bytecode is the same as the bytecode without library. By contrast, the CNN-oriented instruction not only simply provides and encapsulates CNN operations (e.g., Conv, Pooling, etc.), but also includes the optimized computational and data fetching method.

## 4.3 Latency

We consider the inference latency in three aspects: end-to-end latency, weights fetching latency, and inference computing latency (on CPU and GPU platform). The end-to-end latency equals the summary of weights fetching latency and inference computing latency. We run the experimental configuration ten times to avoid random deviation and record the average results. Results of the latency of the four CNN inference are reported in Figs. 12, 13, 14, and 15.

### 4.3.1 *End-to-End Latency*

We product end-to-end latency to show the overall performance of on-chain CNN inference in SmartVM. The end-to-end latency is composed of weights fetching latency and

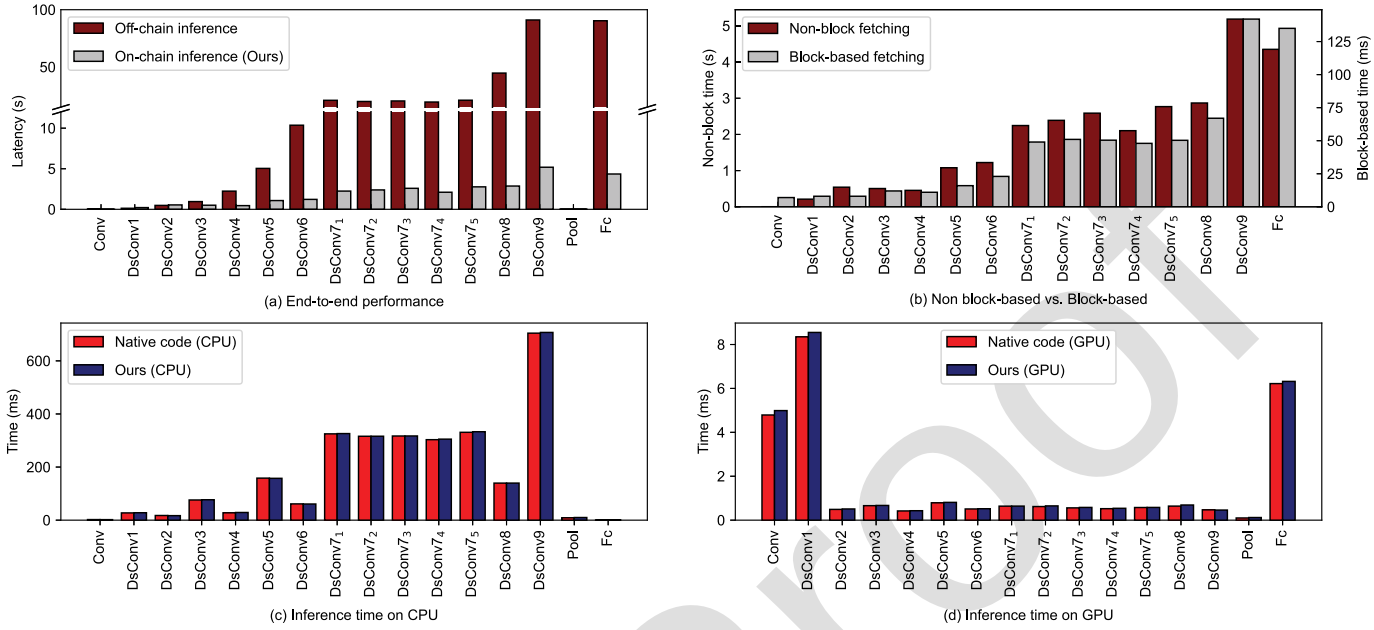



Fig. 15. Results for latency of MobileNet inference.

inference computational latency. We use this metric to show the effectiveness of SmartVM design. The results are given in each first subfigure from Figs. 12, 13, 14, and 15. As the results show, compared with off-chain CNN inference, SmartVM can significantly shorten the overall inference time by 93.6% on average.

The proposed weight prefetching and parallel computation technology also improve the overall performance of SmartVM. The experimental results are given in Table 2. The no pipeline latency is divided into weights fetching latency and computing latency (the total latency equals fetching latency adds computing latency). The pipelined latency is the latency after pipelining the weights fetching and computing. Results show that among the four networks, the pipelined latency is shorter than the total no pipeline time in SmartVM. The results show that with pipeline technology, the end-to-end inference latency can be reduced by 17.8%, 13.7%, 6.7%, and 14.2% on each network, respectively.

In detail, the on-chain inference mode reduces the weights fetching latency by 93.7%, compared with off-chain inference mode. For the four NNs, compared with on-chain inference based on EVM, the CNN-oriented instructions reduce the inference latency by 97.3%, the block-based weights fetching can reduce the inference latency by 98.7%, and with the pipeline computation, the latency can be reduced by 13.1% on average. In addition, SmartVM keeps the pure computation latency similar between the on-chain and off-chain inference on CPU and GPU. As a result, the SmartVM can significantly reduce overall latency by 93.6%.

For the overall latency of LSTM, the weights fetching time is 4867ms and 10ms in the baseline and SmartVM (with block-based weights fetching), respectively. The results for computation time on CPU platform are 7.2ms and 7.4ms by native code and SmartVM, respectively.

TABLE 2
Pipeline Latency

| Network | Latency (ms) | | | |
|---|---|---|---|---|
| | No pipeline | | | Pipelined |
| | Weights fetching | Computing | Total | Total |
| LeNet | 2 | 10 | 12 | 9.87 |
| Alexnet | 1228 | 4370.98 | 5598.98 | 4832.68 |
| ResNet18 | 328 | 7037 | 7365 | 6871 |
| MobileNet | 677.7 | 2826.47 | 3504.17 | 3005.52 |

### 4.3.2 *Weights Fetching Latency*

We give the weights fetching performance evaluation to show the effectiveness of block-based weights fetching technology. In the off-chain CNN inference, the weight fetching refers to invoking smart contract functions to get back all the weights. In the on-chain CNN inference (in SmartVM), the weight fetching refers to invoking contract data trie interface to get weights by CNN-oriented instructions. The results are given in each second subfigure from Figs. 12, 13, 14, and 15. In each figure, the "Non block-based vs. Block-based" is just the time for weight fetching.

As shown, compared with off-chain CNN inference, SmartVM can significantly shorten the weight fetching time by 93.7% on average. Especially, the weight fetching time can be reduced by 98.6% at maximum and 89.6% at minimum. The SmartVM with block-based weight fetching experiment reports a positive time reduction with respect to a solution that executes all CNN inferences on-chain with the default non-block manner. As shown, the speedups with respect to on-chain computing in SmartVM consistently outperform the inference on EVM on-chain without block-based weight fetching depending on the CNN and dataset, SmartVM reduces the weight fetching time by up to 28× (from 57.3ms to 2ms) for LeNet-5 and by up to 210× (from 68901ms to 328ms) for ResNet-18.

With the help of pipeline and parallel computation method, as data is shown in Table 2, the inference latency is

reduced by 17.75%, 13.7%, 6.7%, and 14.2% on LeNet-5, AlexNet, ResNet-18, MobileNet, respectively. The average reduction rate is 13.1%, and the maximum reduction rate is 14.2% on MobileNet except for LeNet-5 (as the number of LeNet-5 weights is too less). In general, the pipelined overall time is longer than non-pipelined computation time, as in some fully-connected layers, the weights fetching time is longer than the computation time. In SmartVM, the proposed pipelined can increase the computation throughput. With the non-pipelined method, the computation process and weights fetching process share the same thread together. With the pipelined method, the computation process and weights fetching process run on the different threads that are created by different Goroutines (concurrency model in Golang). Therefore, the computation time can be reduced with the pipelined method. As a result, in some DNNs with small-scale weights such as LeNet and ResNet, the pipelined overall time is slightly shorter than the pure computation time by 1% to 2%.

From the experimental results, compared with the off-chain inference mode of the BC-AI system, the on-chain inference mode can significantly reduce the weight downloading time, because the function invocation path is shorter. In off-chain mode, downloading one single weight value needs invoking `get()`, `Sload()`, `getStorage()`, and `getTrie()` in order. In on-chain mode, fetching one weight only needs invoking `getStorage()` and `getTrie()`. Moreover, the off-chain mode needs data transmission time from data residence to local platform.

#### 4.3.3 *Inference Computing Latency*

We evaluate the CNN inference pure computing latency to show the effectiveness of the proposed CNN-oriented instructions and parallel computation technology. In the off-chain CNN inference, the computation refers to performing computation from the first layer to the final layer of CNN. In the on-chain CNN inference (in SmartVM), the computation refers to performing the corresponding computation after weights fetching. For example, in `Conv`, the computation latency is the time for convolutional computations after fetching convolutional kernels. The experimental results on CPU and GPU platform are given in the third and fourth subfigures from Figs. 12, 13, 14, and 15, respectively.

According to the computing latency results on CPU and GPU platform, the SmartVM keeps the same latency between native code and on-chain computation. On CPU platform, the computing latency is nearly 10ms, 4.3s, 7s, and 3s for LeNet, AlexNet, ResNet, and MobileNet, respectively. On GPU platform, the computing latency is nearly 1ms, 52ms, 30ms, and 26.3ms for the four networks, respectively. Among the four networks, which are close to the time of CNN inference through the native code.

The SmartVM employs two technologies to reduce inference computing time: SmartVM saves the instruction interpretation time by proposed CNN-oriented instructions, and the block-based method facilitates covering weight fetching time by CNN computation. Firstly, as aforementioned in the code length subsection, the CNN-oriented instructions can reduce 10000 instructions execution in runtime, because the logic of convolutional computation is implemented in the proposed instructions rather than simply putting the original instructions together. Secondly, in the block-based method, the weights are organized by block, and each block is stored in the leaf node of storage trie, while the traditional method stores a single weight in the leaf node of storage trie. Invoking `getTrie()` accesses the trie and returns the value of leave node. So, in order to get all the weights, the `getTrie()` will be invoked many times (e.g., 34,848 times in the first convolutional layer of AlexNet). In the block-based method, invoking once `getTrie()` can return a weight block, which includes many weights. Therefore, prefetching weights by blocks can reduce the number of trie interface invoking (from 34,848 to 96), then the time for invoking trie interface can be decreased by more than 80%. For example, for the large weight data CNN such as AlexNet and ResNet-18, the speedup can be 184x and 210x improved by SmartVM, respectively. It is possible to cover fetching time by computation time in the block-based method as the former is less than the latter. For example, in the first convolutional layer of ResNet, fetching weights needs 11ms, while computation needs 400ms (see Fig. 14). In fact, we observe that in the convolutional layers of four CNNs, the convolutional computation latency can cover weights fetching latency, while in the fully-connected layers, the weights fetching time can cover fully-connected computation latency.

### 4.4 Memory Footprint

We product memory footprint evaluation to show the effectiveness of the proposed dynamic memory management method. We evaluate the physical memory footprint from two aspects: 1) the comparison between the size of EVM Memory and SmartVM Buffer in runtime to show the efficiency of the dynamic memory management method, and 2) the comparison between SmartVM and native code during CNN inference to show that the proposed SmartVM can provide a similar performance compared with native code wise. In our evaluation, the size of each item in Buffer is defined as 256 b, which equals the size of each item in EVM Memory.

The comparison results for EVM Memory and SmartVM Buffer are given in Fig. 16. Results show that with the proposed method, the memory footprint for storing feature maps can be reduced by 84%, 90.8%, 94.3%, and 93.7% on average in LeNet, AlexNet, ResNet18, and MobileNet inference, respectively. In LeNet-5, the peak memory used by Buffer is only 4.2KB of a slice, while this number is more than 2× in EVM Memory. In AlexNet, due to a large number of weights, the memory used by EVM Memory is up to nearly 160MB, while the maximum memory required is only 42MB by SmartVM Buffer. In ResNet18, results show that the minimum and average RAM footprint is 25MB and 61MB in the EVM Memory, respectively. With the help of the management method in SmartVM, the average memory footprint in Buffer can be reduced to only 3.5MB. In MobileNet, the Buffer requires only less than 1MB memory space to perform CNN inference, while the EVM Memory requires more than 4MB space.

For the CNN with a larger number of weights, the efficiency of dynamic memory management method is more significant. For example, in AlexNet, the memory space can be saved by more than 100MB. For the four CNNs, in SmartVM Buffer, the highest memory used is occurred after

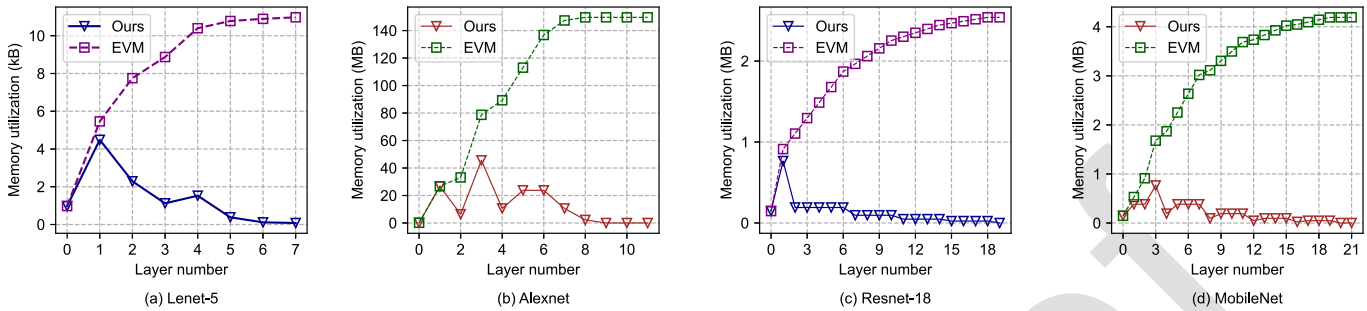



Fig. 16. RAM footprint comparison between SmartVM Buffer and EVM Memory.

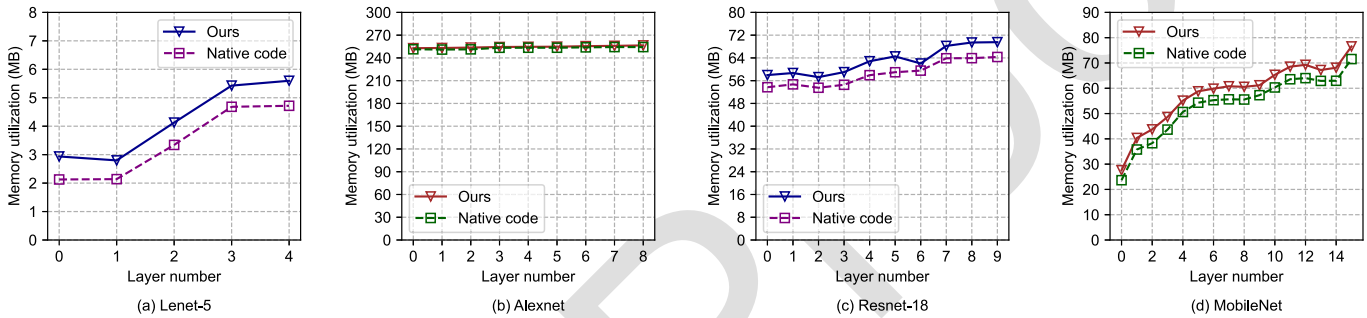



Fig. 17. RAM footprint comparison between SmartVM and native code.

the first layer, because the feature map is the largest of the other layers. In general, the size of EVM Memory is gradually increased because the feature maps will not be freed, while the size of Buffer is decreased because the feature map size is decreased due to the pooling layers.

The memory used results of SmartVM and native code inference comparison is given in Fig. 17. Experimental results show that SmartVM can keep the close latency with slightly higher than native code (6% on average). There are three reasons for the phenomenon: 1) except for trained weights, some block validation-related information (e.g., hash value) has to be stored. 2) In order to support heterogeneous computing, some space for shared libraries (e.g., Cuda) is inevitable, and some space for the interpretation in stack-based SmartVM is required. 3) In order to ensure data consistency and reach consensus accurately and quickly, float computation is not considered in EVM and SmartVM, because the results of float computation are not always the same on a different kind of hardware. In this case, all the runtime data (e.g., including weights, feature maps) are stored as 256 b, while the native code can pick different data width. In summary, the dynamic memory management method can keep a similar memory footprint between SmartVM and native code.

## 4.5 Discussion

In this subsection, we give the analysis of inference accuracy and gas usage to illustrate that the SmartVM can be deployed in intelligent applications and blockchain systems. Lastly, we discuss the potentially optimization for storage space in SmartVM.

### 4.5.1 Accuracy Discussion

The pre-trained model we used is the same between native code and SmartVM, and the experimental results show that the accuracy of image classification in native code and SmartVM can be the same level: 99.98% in LeNet, 57.1% in AlexNet, 69.6% in ResNet18, and 70.9% in MobileNet.

In fact, picking a power of two as the scaling factor also can be implemented in SmartVM, and picking a power of two as the scaling factor is a kind of NN compression and quantization methods such as ESB [47] and TSQ [48]. The quantization methods can further improve the inference performance. The proposed SmartVM focuses on the inference performance without any accuracy loss. However, the model compression and quantization method leads to accuracy loss. Therefore, we pick 1,000 or 10,000 as the scaling factor. Besides, in order to ensure the precision of the computational results and make numerically more stable on different hardware, both the EVM and SmartVM can not support operations of Float type data, so we convert the type of CNN weights from Float to Int by multiplying 1,000 or 10,000. Furthermore, smart contract are computed across different platform and machines, using Int type data for computation can keep a cross platform consistency of results.

### 4.5.2 Gas Usage Discussion

Gas is the virtual unit used in Ethereum to measure the computational and storage resources required to perform certain actions on the Ethereum [49]. For example, ADD instruction costs 2 units of gas, and MUL instruction costs 3 units. We give the on-chain CNN inference gas consumption to prove that the computation amount of inference in SmartVM can be accepted not only in private blockchain but also in public blockchain. We take LeNet-5 as an example, the gas usage during on-chain inference in SmartVM is listed in Table 3. In the three convolutional layers, the results of gas usage are 617556, 222913, and 61146, respectively. In the two pooling layers, the results are 911384 and

TABLE 3
The Gas Usage of LeNet-5 On-Chain Inference

| Layer | Gas usage ($\times 10^4$) | Blocks # in public Ethereum |
|---|---|---|
| Conv1 | 61.8 | #12269695 |
| Pool1 | **91.1** | #12269690 |
| Conv2 | 22.3 | #12269668 |
| Pool2 | 20.8 | #12269670 |
| Conv3 | 6.1 | #12269672 |
| FC1 | 6.1 | #12264715 |
| FC2 | **3.5** | #12264706 |

208038. And in the two fully-connected layers, the results are 61359 and 35047. Note that the computation in Conv1 is larger than in Pool1, but the gas usage is opposite since Pool1 needs to read from EVM Memory more times than Conv1. The gas usage of reading EVM Memory is larger than computation.

In fact, the gas usage can be tolerated in the public Ethereum, because the gas usage in many real blocks is bigger than or similar to the gas usage in the on-chain LeNet-5 inference. Such blocks are also listed in Table 3, the data is fetched from Etherscan.[6]

The gas cost for the proposed CNN-oriented instructions references the Ethereum design. For example, the gas cost for reading Buffer is similar to the reading Memory because the resources consumption is the same. And the gas cost for convolution instruction is generated dynamically according to the computing amount.

#### 4.5.3 Storage Space Discussion

The SmartVM currently focuses on the latency, programming, and gas usage performance during on-chain CNN inference. The blockchain's default storage mechanism, which not only stores original data, but also stores extra data for validation. Such storage mechanism brings a heavy storage burden for blockchain. For example, the weights file size is up to several hundreds of MB, which brings high requirements of distributed storage space, especially for embedded devices. The traditional hash-based data summary needs extra time to find the original off-chain data, so direct data processing methods are needed for the blockchain without data decompression and data decode. It is also difficult to store the temporary data during on-chain CNN inference due to frequent writing and reading operations. Potentially, adopting direct data processing method and text compression technology such as TADOC [50], POCLib [51] and Sequitur [52] is promising for storing the computational temporary data to make the computation more reliable.

In addition to on-chain CNN inference, the blockchain also requires TB-level storage space on hard disk to store the block and transaction data. The POCLib can also be considered to reduce the storage burden of the blockchain itself. In terms of the Etherscan data, the data size in an Ethereum Geth full node is up to 709.2GB in May 17, 2022. During the past three months, the data size increased by about 1GB to 2GB every day. In this case, compressing the blockchain data by some novel techniques such as POCLib is also promising.

6. https://etherscan.io/

## 5 Conclusion

In this paper, we propose SmartVM, which provides architectural support for fast on-chain CNN inference, and enables heterogeneous computing. We present CNN-oriented instruction set to reduce the latency by decreasing the number of instructions in bytecode during CNN on-chain inference. We propose a memory management mechanism to reduce the memory pressure through dynamically space free and allocation according to the size of the feature map. In addition, the weights are stored in the blockchain as blocks, and we organize weights fetching with blocks and computing in a parallel pipeline manner. Experimental results highlight that the inference latency and memory footprint are significantly reduced. Compared with the traditional off-chain computing, SmartVM can speedup the overall execution by 70×, 16×, 11×, and 12× over Lenet-5, Alexnet, Resnet-18, and MobileNet respectively. The memory footprint can be reduced by 84%, 90.8%, 94.3%, and 93.7% over the above four models while offering the same level of accuracy. These results strongly show that SmartVM can be used to promote DNN inference on-chain and be promising to further boost BC-AI applications.

## References


[1] W. Li, Z. Su, R. Li, K. Zhang, and Y. Wang, "Blockchain-based data security for artificial intelligence applications in 6G networks," *IEEE Netw.*, vol. 34, no. 6, pp. 31–37, Nov./Dec. 2020.

[2] M. Mylrea, "AI enabled blockchain smart contracts: Cyber resilient energy infrastructure and IoT," in *Proc. Conf. Assoc. Advance. Artif. Intell. Spring Symp. Ser.*, 2018.

[3] S. Guo, Y. Qi, Y. Jin, W. Li, X. Qiu, and L. Meng, "Endogenous trusted DRL-based service function chain orchestration for IoT," *IEEE Trans. Comput.*, vol. 71, no. 2, pp. 397–406, Feb. 2022.

[4] S. Alrubei, E. Ball, and J. Rigelsford, "The use of blockchain to support distributed ai implementation in IoT systems," *IEEE Internet Things J.*, to be published, doi: 10.1109/JIOT.2021.3064176.

[5] M. Keshk, B. Turnbull, N. Moustafa, D. Vatsalan, and K.-K. R. Choo, "A privacy-preserving-framework-based blockchain and deep learning for protecting smart power networks," *IEEE Trans. Ind. Informat.*, vol. 16, no. 8, pp. 5110–5118, Aug. 2020.

[6] T. Bui *et al.*, "ARCHANGEL: Tamper-proofing video archives using temporal content hashes on the blockchain," in *Proc. Conf. Comput. Vis. Pattern Recognit. Workshops*, 2019, pp. 2793–2801.

[7] X. Lin, J. Li, J. Wu, H. Liang, and W. Yang, "Making knowledge tradable in edge-AI enabled IoT: A consortium blockchain-based efficient and incentive approach," *IEEE Trans. Ind. Informat.*, vol. 15, no. 12, pp. 6367–6378, Dec. 2019.

[8] A. Goel, A. Agarwal, M. Vatsa, R. Singh, and N. Ratha, "DeepRing: Protecting deep neural network with blockchain," in *Proc. Conf. Comput. Vis. Pattern Recognit. Workshops*, 2019, pp. 2821–2828.

[9] I. Sergey, V. Nagaraj, J. Johannsen, A. Kumar, A. Trunov, and K. C. G. Hao, "Safer smart contract programming with scilla," *Proc. ACM Program. Lang.*, vol. 3, pp. 1–30, 2019.

[10] J. Chen, X. Xia, D. Lo, J. Grundy, X. Luo, and T. Chen, "Defining smart contract defects on ethereum," *IEEE Trans. Softw. Eng.*, vol. 48, no. 1, pp. 327–345, Jan. 2022.

[11] N. Grech, M. Kong, A. Jurisevic, L. Brent, B. Scholz, and Y. Smaragdakis, "MadMax: Surviving out-of-gas conditions in ethereum smart contracts," *Proc. ACM Program. Lang.*, vol. 2, pp. 1–27, 2018.

[12] T. Li, Y. Fang, Z. Jian, X. Xie, Y. Lu, and G. Wang, "ATOM: Architectural support and optimization mechanism for smart contract fast update and execution in blockchain-based IoT," *IEEE Internet Things J.*, vol. 9, no. 11, pp. 7959–7971, Jun. 2022.

[13] X. Larrucea and C. Pautasso, "Blockchain and smart contract engineering," *IEEE Softw.*, vol. 37, no. 5, pp. 23–29, Sep./Oct. 2020.

[14] J. Eberhardt and S. Tai, "On or off the blockchain? Insights on off-chaining computation and data," in *Proc. Eur. Conf. Serv.-Oriented Cloud Comput.*, 2017, pp. 3–15.

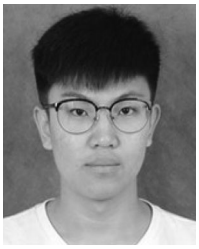
[15] L. Su *et al.*, "Evil under the sun: Understanding and discovering attacks on ethereum decentralized applications," in *Proc. 30th USENIX Secur. Symp.*, 2021, pp. 1307–1324.
[16] M. Yano, C. Dai, K. Masuda, and Y. Kishimoto, *Blockchain and Crypto Currency: Building a High Quality Marketplace for Crypt Data*. Berlin, Germany: Springer, 2020.
[17] Z. Zheng *et al.*, "Agatha: Smart contract for DNN computation," 2021, *arXiv:2105.04919*.
[18] Y. Yang, "Training massive deep neural networks in a smart contract: A new hope," 2021, *arXiv:2106.14763*.
[19] Z. Hu, B. Li, and J. Luo, "Time-and cost-efficient task scheduling across geo-distributed data centers," *IEEE Trans. Parallel Distrib. Syst.*, vol. 29, no. 3, pp. 705–718, Mar. 2018.
[20] A. C. Zhou, Y. Xiao, Y. Gong, B. He, J. Zhai, and R. Mao, "Privacy regulation aware process mapping in geo-distributed cloud data centers," *IEEE Trans. Parallel Distrib. Syst.*, vol. 30, no. 8, pp. 1872–1888, Aug. 2019.
[21] W. Zou *et al.*, "Smart contract development: Challenges and opportunities," *IEEE Trans. Softw. Eng.*, vol. 47, no. 10, pp. 2084–2106, Oct. 2019.
[22] J.-Y. Kim and S.-M. Moon, "Blockchain-based edge computing for deep neural network applications," in *Proc. Workshop Intell. Embedded Syst. Archit. Appl.*, 2018, pp. 53–55.
[23] G. Wood *et al.*, "Ethereum: A secure decentralised generalised transaction ledger," *Ethereum Project Yellow Paper*, vol. 151, no. 2014, pp. 1–32, 2014.
[24] T. Dickerson, P. Gazzillo, M. Herlihy, and E. Koskinen, "Adding concurrency to smart contracts," *Distrib. Comput.*, vol. 33, no. 3, pp. 209–225, 2020.
[25] S. Wang, L. Ouyang, Y. Yuan, X. Ni, X. Han, and F.-Y. Wang, "Blockchain-enabled smart contracts: Architecture, applications, and future trends," *IEEE Trans. Syst., Man, Cybern. Syst.*, vol. 49, no. 11, pp. 2266–2277, Nov. 2019.
[26] Q. Fan, D.-P. Fan, H. Fu, C.-K. Tang, L. Shao, and Y.-W. Tai, "Group collaborative learning for co-salient object detection," in *Proc. Conf. Comput. Vis. Pattern Recognit.*, 2021, pp. 12288–12298.
[27] Y. Chen, J. He, X. Zhang, C. Hao, and D. Chen, "Cloud-DNN: An open framework for mapping DNN models to cloud FPGAs," in *Proc. ACM/SIGDA Int. Symp. Field-Program. Gate Arrays*, 2019, pp. 73–82.
[28] Y. Lecun, L. Bottou, Y. Bengio, and P. Haffner, "Gradient-based learning applied to document recognition," *Proc. IEEE*, vol. 86, no. 11, pp. 2278–2324, Nov. 1998.
[29] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," in *Proc. Conf. Comput. Vis. Pattern Recognit.*, 2016, pp. 770–778.
[30] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "Imagenet classification with deep convolutional neural networks," in *Proc. Adv. Neural Inf. Process. Syst.*, 2012, pp. 1097–1105.
[31] G. Huang, Z. Liu, L. Van Der Maaten, and K. Q. Weinberger, "Densely connected convolutional networks," in *Proc. Conf. Comput. Vis. Pattern Recognit.*, 2017, pp. 4700–4708.
[32] M. Lin *et al.*, "HRank: Filter pruning using high-rank feature map," in *Proc. Conf. Comput. Vis. Pattern Recognit.*, 2020, pp. 1529–1538.
[33] Y. Wang, C. Xu, C. Xu, and D. Tao, "Beyond filters: Compact feature map for portable deep model," in *Int. Conf. Mach. Learn.*, 2017, pp. 3703–3711.
[34] S. Lin *et al.*, "Towards optimal structured cnn pruning via generative adversarial learning," in *Proc. Conf. Comput. Vis. Pattern Recognit.*, 2019, pp. 2790–2799.
[35] K. Simonyan and A. Zisserman, "Very deep convolutional networks for large-scale image recognition," 2014, *arXiv:1409.1556*.
[36] A. Dosovitskiy *et al.*, "An image is worth 16x16 words: Transformers for image recognition at scale," in *9th Proc. Int. Conf. Learn. Representations*, 2021.
[37] P. Tolmach, Y. Li, S.-W. Lin, Y. Liu, and Z. Li, "A survey of smart contract formal specification and verification," *ACM Comput. Surv.*, vol. 54, no. 7, pp. 1–38, 2021.
[38] C. Xu *et al.*, "Making Big Data open in edges: A resource-efficient blockchain-based approach," *IEEE Trans. Parallel Distrib. Syst.*, vol. 30, no. 4, pp. 870–882, Apr. 2019.
[39] M. Shen, Y. Deng, L. Zhu, X. Du, and N. Guizani, "Privacy-preserving image retrieval for medical IoT systems: A blockchain-based approach," *IEEE Netw.*, vol. 33, no. 5, pp. 27–33, Sep./Oct. 2019.
[40] T. D. Nguyen, L. H. Pham, J. Sun, Y. Lin, and Q. T. Minh, "sFuzz: An efficient adaptive fuzzer for solidity smart contracts," in *Proc. ACM/IEEE 42nd Int. Conf. Softw. Eng.*, 2020, pp. 778–788.
[41] K. Wüst, S. Matetic, S. Egli, K. Kostiainen, and S. Capkun, "ACE: Asynchronous and concurrent execution of complex smart contracts," in *Proc. ACM SIGSAC Conf. Comput. Commun. Secur.*, 2020, pp. 587–600.
[42] N. He *et al.*, "EOSAFE: Security analysis of EOSIO smart contracts," in *Proc. 30th USENIX Secur. Symp.*, 2021, pp. 1271–1288.
[43] Q. Xiao, Y. Liang, L. Lu, S. Yan, and Y.-W. Tai, "Exploring heterogeneous algorithms for accelerating deep convolutional neural networks on FPGAs," in *Proc. 54th Annu. Des. Automat. Conf.*, 2017, pp. 1–6.
[44] C. Zhang, P. Li, G. Sun, Y. Guan, B. Xiao, and J. Cong, "Optimizing FPGA-based accelerator design for deep convolutional neural networks," in *Proc. ACM/SIGDA Int. Symp. Field-Program. Gate Arrays*, 2015, pp. 161–170.
[45] J. Deng and W. E. A. Dong, "ImageNet: A large-scale hierarchical image database," in *Proc. IEEE Conf. Comput. Vis. Pattern Recognit.*, 2009, pp. 248–255.
[46] Y. Chen *et al.*, "An instruction set architecture for machine learning," *ACM Trans. Comput. Syst.*, vol. 36, no. 3, Aug. 2019.
[47] C. Gong, Y. Lu, K. Xie, Z. Jin, T. Li, and Y. Wang, "Elastic significant bit quantization and acceleration for deep neural networks," *IEEE Trans. Parallel Distrib. Syst.*, to be published, doi: 10.1109/TPDS.2021.3129615.
[48] P. Wang, Q. Hu, Y. Zhang, C. Zhang, Y. Liu, and J. Cheng, "Two-step quantization for low-bit neural networks," in *Proc. IEEE Conf. Comput. Vis. Pattern Recognit.*, 2018, pp. 4376–4384.
[49] X. L. Yu, O. Al-Bataineh, D. Lo, and A. Roychoudhury, "Smart contract repair," *ACM Trans. Softw. Eng. Methodol.*, vol. 29, no. 4, pp. 1–32, 2020.
[50] F. Zhang *et al.*, "TADOC: Text analytics directly on compression," *VLDB J.*, vol. 30, no. 2, pp. 163–188, 2021.
[51] F. Zhang, J. Zhai, X. Shen, O. Mutlu, and X. Du, "POCLib: A high-performance framework for enabling near orthogonal processing on compression," *IEEE Trans. Parallel Distrib. Syst.*, vol. 33, no. 2, pp. 459–475, Feb. 2022.
[52] C. G. Nevill-Manning and I. H. Witten, "Identifying hierarchical structure in sequences: A linear-time algorithm," *J. Artif. Intell. Res.*, vol. 7, pp. 67–82, 1997.

**Tao Li** received the PhD degree in computer science from Nankai University, China in 2007. He currently works with the College of Computer Science, Nankai University as a professor. He is the member of the IEEE Computer Society and the ACM, and the distinguished member of the CCF. His main research interests include heterogeneous computing, machine learning, and Internet of Things.

**Yaozheng Fang** received the BS degree from the Hebei University of Technology, Tianjin, China in 2019. He is currently working toward the PhD degree with the College of Computer Science, Nankai University. His main research interests include blockchain, smart contract, and Internet of Things.

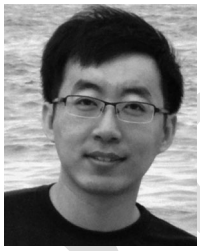
**Ye Lu** received the BS and PhD degrees from Nankai University, Tianjin, China in 2010 and 2015, respectively. He is currently working as an associate professor with the College of Cyber Science, Nankai University. His main research interests include DNN FPGA accelerator, blockchian virtual machine, embedded system, Internet of Things.

**Jinni Yang** received the BEng degree in Internet of Things from Nankai University in 2020. She is currently working toward the master's degree in computer science with Nankai University. Her main research is in blockchain security.



**Zhaolong Jian** received the BEng degree in Internet of Things from Nankai University in 2020. He is currently working toward the MSc degree with the College of Computer Science, Nankai University. His main research interests include smart contract virtual machine, blockchain system security, and Internet of Things.

**Zhiguo Wan** received the BS degree in computer science from Tsinghua University, Beijing, China, in 2002, and the PhD degree from the School of Computing, National University of Singapore, Singapore, in 2007. He is currently working as an associate professor with the School of Computer Science and Technology, Shandong University, Jinan, China. He worked as a postdoctoral fellow with the Katholieke University of Leuven, Leuven, Belgium, from 2006 to 2008. His main research interests include security and privacy for Big Data, cryptocurrency, and blockchain.



**Yusen Li** received the PhD degree in computer science from Nanyang Technological University, in 2013. He is currently working as an associate professor with the Department of Computer Science, Nankai University, China. His research interests include resource allocation and scheduling issues in distributed systems and cloud computing.