AKGF: Automatic Kernel Generation for DNN on CPU-FPGA

Dong Dong1,*, Hongxu Jiang1,2* and Boyu Diaó2,3*

1Beijing Key Laboratory of Digital Media, State Key Lab Virtual Real Technology and Systems, Beihang University, Beijing, 100191, China
2Institute of Computing Technology, Chinese Academy of Sciences, Beijing, 100086, China
3Corresponding author: BY1906006@buaa.edu.cn, jianghx@buaa.edu.cn, diaoboyu2012@ict.ac.cn

While tensor accelerated compilers have proven effective in deploying deep neural networks (DNN) on general-purpose hardware, optimizing for FPGA remains challenging due to the complex DNN architectures and the heterogeneous, semi-open compute units. This paper introduces the Automatic Kernel Generation for DNN on CPU-FPGA (AKGF) framework for efficient deployment of DNN on heterogeneous CPU-FPGA platforms. AKGF generates an intermediate representation (IR) of the DNN using TVM’s Halide IR, annotates the operators of model layers in the IR to compute them on the corresponding hardware cores, and further optimizes the operator code for CPU and FPGA using ARM’s function library and the polyhedral model to enhance model inference speed and power consumption. The experimental tests conducted on a CPU-FPGA board validate the effectiveness of AKGF, demonstrating significant acceleration ratios (up to 6.7x) compared to state-of-the-art accelerators while achieving a 2x power optimization. AKGF effectively leverages the computational capabilities of both CPU and FPGA for high-performance deployment of DNN on CPU-FPGA platforms.

Keywords: DNN accelerated compilers; polyhedral model; heterogeneous computing; CPU-FPGA

1. INTRODUCTION

Due to the high pixel count of images and complex algorithms, many deep neural networks (DNN) have optimization bottlenecks in terms of computational and energy efficiency. Considering that there is a lot of data locality and parallelism in these applications, we can take advantage of these characteristics to improve the performance of the code through optimized execution strategies, instead of manual code tuning work. However, this greatly increases coding complexity and requires the programmer to have knowledge of hardware programming languages. So developing a domain-specific language (DSL), such as Halide [1], is an effective way to reduce the amount of detailed knowledge required for application creation [2]. In the presence of such code optimization tools, the use of heterogeneous CPU-FPGA can surpass the computational performance of high-power programmable architectures and fixed-function energy-efficient ASICs [3]. Heterogeneous CPU-FPGA reconfiguration capabilities can generate high-performance, low-power deep neural network hardware mapping code that meets system-level requirements for throughput, latency and power in a variety of environments from embedded systems.

For the FPGA platform, the polyhedral model is used for compilation optimization acceleration [4]. The polyhedral model is a compilation framework for circular transformations [5]. Most previous pulsation array compilers were built on polyhedral frameworks. What these frameworks have in common is the use of spatial-temporal transformations to transform algorithms into new programs that describe the architecture of systolic arrays. These compilers are comprehensively automated, improving model efficiency, but most of them do not provide performance comparable to manual designs because they miss some hardware optimization methods that help to improve computing and communication efficiency, such as memory resources, architecture, power, etc.

In previous work, polyhedral models are commonly used in compilers, one of which is the Integer Set Library (ISL) [6]. Polyhedral type compilers that support automatic scheduling (non-scheduling languages) include Pluto [7], PENCIL [8], Polly [9], Tensor Comprehensions [10], PolyMage [11]; those that provide scheduling languages include Tiramisu [12], AlphaZ [13], CHiLL [14], PolySA [15]. Tensor comprehensions are often used as baselines for the comparison of deep learning compilers, which are compilers based on black-box automatic tuning algorithms. PolyMage is a compiler for automatic image processing. Pluto can automatically schedule algorithms and is also used in PENCIL and Polly. Its scheduling algorithm wants to shorten the distance between producer and consumer expressions while maximizing the outermost parallelism. But the data layout, redundant computation and control complexity of the generated code are not considered. Tiramisu is a compiler that absorbs the advantages of the polyhedral model. Although it does not support automatic scheduling, it provides a scheduling language that allows full control of the scheduling process. In addition, compared to the above polyhedral model compiler, it also implements many optimization methods, such as data packing, data prefetching, memory partitioning and asynchronous data communication.

Halide is the primary compiler for this approach. It is a DSL for deep neural network processing. Halide supports scheduling languages. Halide-based compilers such as Distributed Halide [16], support Halide for distributed computing. TVM [17] is also
based on Halide, and of course, some improvements have been made later. The VTA [18] in TVM provides FPGA-oriented deployment tools, but it needs to rely on the corresponding intellectual property (IP) core and cannot optimize the code itself. Hetero-Halide [19] is a neural network accelerator dedicated to compiling Halide programs to FPGA while optimizing the code using specific optimization algorithms and scheduling information in the Halide program. However, this method does not perform block optimization operations for the neural network structure, resulting in that the CPU cores in the heterogeneous FPGA do not exert their computing advantages and cannot exert the maximum computing performance of different cores.

In this paper, we propose a novel approach for partitioning deep models between the CPU and FPGA to exploit their respective strengths. The methodology involves analyzing the computational requirements of different layers in the model and determining the most suitable device for executing each layer. We also introduce a core allocation strategy that dynamically assigns computational tasks to the available CPU and FPGA cores, considering factors such as load balancing and resource utilization. Our approach requires access to the FPGA’s official code generation tool (which currently implements automatic code optimization and generation for Xilinx). Our first priority is to improve versatility by compiling any program as a pulsating array, as long as they are supported by a polyhedral framework and can legally map to pulsating arrays. To compile such programs into pulsation arrays, we propose techniques and optimizations for automatically converting conventional sequential programs into parallel programs that describe a complete pulsation array system, including processing elements (PE) and on-chip I/O networks. Secondly, Automatic Kernel Generation for DNN on CPU-FPGA (AKGF) groups DNN for heterogeneous cores to maximize the performance of operators. By parsing and labeling the code of the model layer, we add corresponding labels to the code sentences and manage the code operation strategy uniformly through the array. The CPU and FPGA can process the corresponding model statements. In this paper, we show that AKGF can support applications with complex and irregular dependent structures, such as GEMM[20], AlexNet[21], VGGNet[22] and YOLO[23].

2. RELATED WORK AND MOTIVATION

Polyhedral compilers: PolySA covered the largest set of optimization techniques. However, the versatility of PolySA is limited because it only supports programs with a single statement in a perfectly nested loop. In addition, there is a limited implementation of PolySA, which is built on Matlab and is not scalable in dealing with complex designs.

TVM: VTA is a parametric accelerator that can accelerate a large number of deep learning computations. The compiler uses a two-level programming interface to explicitly program the VTA. The parameters of the architecture include the size of the GEMM core, the shape of the SRAM, and the data bit width. The benefit of parametric hardware architecture is that the same design can be ported to devices with different hardware resources. VTA consists of four modules: fetch, load, compute, and store. These modules as a whole constitute a task pipeline, which has high computing resource utilization in computing-constrained scenarios and high memory bandwidth utilization in storage-constrained scenarios. However, for the execution pipeline of DNN of arbitrary depth, the circular dependencies are not handled very well.

Halide: HeteroHalide is a neural network accelerator dedicated to compiling Halide programs to FPGA while optimizing the code using specific optimization algorithms and scheduling information in the Halide program. HeteroHalide not only simplifies code migration but also enables efficient accelerator design through its flexible code generation backend module. However, this method does not perform block optimization operations for the neural network structure, resulting in that the CPU cores in the heterogeneous FPGA do not exert their computing advantages and cannot exert the maximum computing performance of different cores.

3. DESIGN AND IMPLEMENTATION

3.1. Tools

In this paper, TVM is used to convert the models from different frameworks to form ONNX files. Halide IR is converted into polyhedral IR under TVM IR, and the optimized code is generated for the target hardware through a series of optimization methods such as code grouping, loop optimization, loop block and automatic tuning.

IR: Halide IR is an intermediate representation used to connect the Halide source code and the corresponding object code of the CPU-FPGA architecture. Halide first reduces the source code to the IR level, then converts it to the IR of the polyhedron model for a series of optimization operations, and then connects to the corresponding hardware object code generator to generate the final code. We use polyhedral IR to analyze the syntax and semantics of Halide programs of algorithmic models and convert them into abstract syntax trees (AST). Each node in the AST represents operations and scheduling operations on variables, such as matrix multiplication and accessing data. In the process of analyzing the grammar, we found that the vast majority of nodes point to other nodes. Therefore, we build this AST and parse out nodes that can be optimized using polyhedral model tools to establish connections between operations and variables, and use the optimized AST as IR in AKGF.

Polyhedral model: The idea of the polyhedral model algorithm is to treat multi-layer for loop as multi-dimensional space. Through static analysis and compiler optimization algorithm, the
3.3. Legality check and grouping

We propose a method for grouping DNN code legality for CPU-FPGA heterogeneous platforms. We use halide IR in TVM to convert the model to polyhedral IR and check the model compliance after getting the polyhedral IR.

In order to legalize the code-to-systolic array mapping, the loop transformation should satisfy the following constraints: First, the transformation should preserve the code semantics. Second, the spatial dependence distance of each cycle should not be greater than 1, in order to make data communication only occur between adjacent PE. We only examine data flows related to inter-PE communication and read cyclic dependencies. Since each PE has its own address space, input dependencies and output dependencies have no effect on data communication between PE. Thirdly, we will parse the code block involving the nonlinear function in the statement, and label the beginning and end of the code, specifying that this part is allocated to CPU computing, and specifying the data transfer method between cores.

3.4. CODE-PE loop transformation

We propose a CODE-PE loop transformation method to adapt the loop structure to the systolic array in order to better utilize the parallelism of FPGA. The specific algorithm is shown in Algorithm 1.

![Figure 2. AKGF architecture.](image)

### Algorithm 1: CODE-PE loop transformation

<table>
<thead>
<tr>
<th>Data:</th>
<th>A schedule tree</th>
</tr>
</thead>
<tbody>
<tr>
<td>Result:</td>
<td>A list of schedule trees</td>
</tr>
<tr>
<td>1</td>
<td>Initialize the space loop candidate pool;</td>
</tr>
<tr>
<td>2</td>
<td>Extract the outermost permutable loop band from;</td>
</tr>
<tr>
<td>3</td>
<td>for each loop in the band do</td>
</tr>
<tr>
<td>4</td>
<td>if all flow/read dependence distances on loop then</td>
</tr>
<tr>
<td>5</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>end</td>
</tr>
<tr>
<td>7</td>
<td>end</td>
</tr>
<tr>
<td>/* Generate 2D systolic array */</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>for each pair of loops in the pool P do</td>
</tr>
<tr>
<td>9</td>
<td>Duplicate the schedule tree s;</td>
</tr>
<tr>
<td>10</td>
<td>Modify s’ by permuting the loops to outermost;</td>
</tr>
<tr>
<td>11</td>
<td>S’</td>
</tr>
<tr>
<td>12</td>
<td>end</td>
</tr>
<tr>
<td>/* Generate 1D systolic array (omitted), similar to 2D case with only one space loop selected */</td>
<td></td>
</tr>
</tbody>
</table>

According to the results of the data dependency analysis, loop groups whose internal computations can be parallelized are screened out as candidate outer loops. The number of layers of the outer loop is selected according to the specific structure of the FPGA. The rest of the content acts as an inner loop, executing sequentially in the systolic array. The candidate outer loops are used in turn to build the scheduling tree. Finally, a series of scheduling trees are generated as possible optimal schedules.

3.5. Code linking

More and more DNN use backbone and follow-up to independently design the structure of the network. For the backbone, it is really not suitable for FPGA, and it is more suitable for the CPU to calculate. But for the network behind the backbone, the fixed $3 \times 3$ convolution, $1 \times 1$ convolution, and batch normalization in the network structure are run by FPGA. In this step, we propose a method that can mark the backbone and activation function of
DNN and store the marks in an array. Using arrays to guide data flow allocation, different model layers are assigned to different cores for computation. In order to avoid the confusion of code running, the sequence number marked in the array is verified by the correct compilation process in the model calculation diagram to determine the correctness of the running code.

### 3.6. Loop fusion for locality

Loop fusion can enhance data locality by reducing the time between uses of the same data, thereby increasing the probability that the data will remain in the cache, resulting in efficient code execution. We found that the arrays in the program of DNN are usually large arrays. Let us take a simple two-round array traversal algorithm as an example (as shown in Fig. 3(a)).

This way of writing has the problem of data locality. It can be seen that in the first loop, the array elements $A(1) ... A(N)$ are written in sequence, and in the second loop, array A is read immediately. The problem here is that when $N$ is a very large (large array), $A(1)$ is already out of the cache when $A(N)$ is written, so it needs to be reloaded into the cache for subsequent reads.

When we merge the two loops (as shown in Fig. 3(b)), the written value is read in time, so the cache miss can be reduced by half. For any loop optimization, we need to focus on its legitimacy (without changing program semantics) and its benefits. Let us first discuss the legality of loop merging. Judging the legitimacy of the merger actually requires systematic dependency analysis. We use some intuitive examples to illustrate the dependencies between data (as shown in Fig. 3(c)).

It is illegal to merge the above two loops because $A(i + 1)$ in $S2$ will read the value written by the first loop before merging, and $A(i + 1)$ will read the value after merging Modify the previous value, so the data dependency is broken. We can do this with loop alignment (as shown in Fig. 3(d)). It is illegal to merge the above two loops because $A(i + 1)$ in $S2$ will read the value written by the first loop before merging, and $A(i + 1)$ will read the value after merging Modify the previous value, so the data dependency is broken. We can do this with loop alignment (as shown in Fig. 3(d)). We first increase the index of the second loop by one, so that the original $A(i + 1)$ can be changed to $A(i)$, so that if merged, $S2$ will read the new value.

For FPGA, we refine the fusion strategy according to the memory level of the code. The data relationships produced by our inverse strategy can be used to instantiate extended model nodes for fusion after tensor data tiling, which is innovative in the classical polyhedron framework. Local loop fusion algorithms manage multi-directional data flow, requiring different fusion strategies used when offloading data to the chip. Tile data accessed by different compute units should be separated, with some passed to L1 and some passed to L2. When the operator is a matrix product, it is passed to the Cube Unit; the others are handled by the scalar unit.

### 3.7. Loop tiling for locality

Loop tiling is one of the most important optimizations to improve data locality, and it is a top priority for both traditional and deep compilers. Due to limited on-chip resources, array partitioning is necessary when mapping the model’s data array to the FPGA. To achieve this, the AKGF tiles the outermost permutable loop strip in the scheduling tree containing the spatial loop. According to the mainstream model characteristics, this algorithm uses the tiling factor of $(4, 4, 4)$ to tile the data of the outermost annulus of the model. And given a loop nest, AKGF can automatically determine whether the loop block is feasible, and if it is feasible, select the most suitable loop layer to block to achieve the maximum benefit.

This section discusses some more complex nested loops. There are several ways to make loops complicated: (i) more levels of nesting, (ii) there is a dependency between the bounds of different loop layers, and there are data dependencies between loops.

For any loop compilation, we need to discuss its legality and profitability. Legality means that it does not change the semantics of the program. In order to ensure the legality of the loop, we share the nested dependencies of the loop as much as possible, thus using fewer cache resources when deployed on the hardware.

Let’s look at a general matrix multiplication three-layer loop as shown in Fig. 4(a): The I-layer loop has been moved to the innermost layer for better spatial locality, which is one of the most common optimizations for matrix multiplication. Note that any interchange is legal for this loop, so tiling is also legal.

It should be pointed out that the transpose matrix itself has a lot of overhead, and after the block, it can actually achieve the same cache hit rate as the transpose (one is a temporal locality and the other is a spatial locality). And $B$ in code may be accessed line by line later, which means that the data layout may not be changed. That is why partitioning becomes very important.

About reuse patterns, let us start with the K-layer loop to see if any access is reused in the K-layer loop. $C(i, J)$ is reused in the K-layer loop, and after tiling the innermost $I$-loop, it will loop in the K-layer. In this case, $C(i, J)$ can be reused.

Let’s look at the outermost $J$-layer loop. $A(i, K)$ is reused in the J-layer loop. When the K and I-layer loops are tiled, we can reuse $A(i, K)$ in the J-layer loop. So, the result after tiling the I and K-layers in turn is shown in Fig. 4(b) (ENDDO is omitted).

Because I was the innermost loop, the reuse of $B(K, J)$ in each round of $I$ was logical, that is, it can be reused without tiling. And now after I and K are tiled, I is moved to the outer loop, so at this time, the reuse of $B(kk, j)$ in the outer I-loop (of course, the inner ii-loop is also logically reused), so we can also loop through the tile $J$-layer to get this new reuse. We can calculate that the cache miss of $B(kk, j)$ is $\frac{I}{S} \times N \times \frac{K}{T} \times \frac{J}{S} = \frac{N}{B}$, which is improved. We can see that $C(i, ii, jj)$ is only reused in the memory $kk$-loop, but not in the outer K-loop. If there is a larger cache, the outer K-loop can also reuse $C(ii, jj)$. This is the tiling of AKGF’s multi-level cache.

As shown in Fig. 5(b), the three-layer loops of ii, kk, and $jj$, can fit in the L1 cache (tile size is T1), and then the I-layer loop cannot fit in L1 but can fit in the L2 cache (tile size is T2). So at this time,
the reuse of the L-layer data in the K-layer loop in the L2 cache is formed, such as \( C(i, j) \). Similarly, we can also do L3 cache tiling.

3.8. Latency hiding

The latency hiding of AKGF is essentially a way to eliminate the on-chip pipeline stalls caused by dependencies carried by computational statements. For example, each multiply-add operation in a loop depends on the previous one in each loop, resulting in a very long chain of dependencies carried by the loop. To solve this problem, AKGF looks for parallel loops in the scheduling tree, strip-digs them and permutes the innermost point loop. For example, we strip mine them with a tiling factor of \((4, 4)\) and permute the innermost point loop. Since there is no circular dependency in the innermost loop, the data throughput rate for FE can be improved.

AKGF memory latency hiding is one of the optimizations considered by TVM. However, TVM does not consider heterogeneous computing units, resulting in insufficient low-level code optimization. We first group the code using a grouping labeling strategy, and then insert low-level code optimization strategies into the generated code through data dependencies across different computing units, maximizing the computational power of the on-chip CPU and FPGA.

3.9. Auto-tuner and code generation

Auto-tuner: The polyhedral model optimization provides a huge solution space, and the system needs to search the solution space to find the schedule with the best actual performance. At present, most deep learning compilers have their own automatic tuning module. The difference between the automatic tuning module proposed in this paper is that we consider the code division of CPU and FPGA, which means that the solution space will not only contain the set of scheduling trees but also include different code divisions.

This module is based on TVM’s automatic scheduler. There are several key components to this entire tuning process. The task scheduler is responsible for task scheduling. As mentioned earlier, the entire network is divided into multiple subgraphs, and each subgraph corresponds to a task. According to the principle of legal code grouping, some special loop calculations and nonlinear functions are separated, and the scheduler is responsible for scheduling these tasks so that those tasks that are most likely to improve performance can be tuned first. For each subgraph, the sketch generation process generates a sketch. It can be understood as relatively coarse-grained scheduling, in which the general loop structure is determined. AKGF then produces more fine-grained optimizations (involving loop fusion, loop tiling, unrolling, etc.), resulting in a complete schedule. It mainly provides the initial population for the next search. Finally, the search is performed by the Evolutionary algorithm. In order to reduce time-consuming performance testing, the search process also trains the cost model for performance evaluation of the generated programs. According to the evaluation results, candidates with high scores are selected for performance testing. Then, the performance data measured by running on real hardware are fed to the cost model for training to improve its accuracy, and this cycle is repeated.

The main function of the cost model in AKGF is to evaluate the time and power consumption of each operator running on the target hardware platform (CPU-FPGA) without actual deployment and to provide a benchmark for dynamic fusion and segmentation of computational graph. In order to achieve the automatic optimization of the target hardware platform perception, it is necessary to build a new most appropriate cost model. The cost model using the deep learning method requires a large amount of data during construction, and it takes quite a long time to obtain data in actual use. In order to speed up the optimization process, the amount of data must be reduced. Therefore, the semi-supervised regression model is used as the benchmark model.

First, extract the corresponding pre-training model according to CPU-FPGA architectures. By traversing the computational graph, AKGF extracts the features of all operators (the fusion operator and the operators that make it up), including the size of the input data, the type of the input data, the number of operators that make up the fusion operator and the type of operation. At the same time, AKGF will also dynamically adjust the representation form of the computational graph according to the information of the hardware to be deployed. Hardware information mainly includes supported parallelism, supported memory transactions (such as 8 bytes, 32 bytes read and write), shared memory restrictions, register restrictions, etc. Then AKGF selects 10% operators to deploy on the CPU-FPGA, and the actual measurement time and power consumption are used as label data. According to the semi-supervised algorithm, the label data is used to build the model and predict the unlabeled data. Finally AKGF adds high confidence data to the labeled dataset, uses it as labeled data and retrains the model. When more than a certain proportion of data in the data set is marked, the training ends and a trained cost model is obtained.

In order to describe the execution process of the cost model in more detail, this paper gives Algorithm 2: Multilevel Rule Guided Cost Model Construction.

The input of the algorithm is the operator list \( P \) to be tested and a parameter \( AC \) related to the model accuracy, and finally outputs a trained cost model. Line 1 loads the pre-trained model, and Line 2 randomly shuffles the input operator list and takes out the top ones as part of preparing to obtain runtime data. Line 4 line10 is the process of model training. Firstly, obtain the actual inference time, power consumption or other required indicators of the operators in the list and then add them...
to the data label set. A cost model is trained on labeled data and then used to predict other unlabeled data. Take out the operators with the highest predicted performance and prepare for the next round of actual testing. When the proportion of labeled data to the overall data exceeds AC, the construction of the cost model ends.

**Code Generation:** After the above steps, AKGF generates an AST from the optimized program, and then performs a build operation to traverse the AST to generate the final design for the CPU-FPGA.

### 3.10. Nonlinear function

The CPU is used to obtain network model weight parameters and input feature map data, and to process some nonlinear functions, such as Sigmoid, tanh, and exponential linear units. The CPU sends process control instructions to the FPGA. The FPGA reads the network model weight parameters and input feature map data from the CPU side, performs code optimization calculations and finally feeds back the intermediate results to the CPU. The code achieves final code generation through HLS.

### 4. EVALUATION

#### 4.1. Experimental Setup

For a given DNN, we use AKGF to obtain the best execution of the code to be deployed. Import the ONNX format files of different models into AKGF, and finally generate the HLS source code. In this article, we use Vivado HLS (v2022.1) for C code compilation and logic simulation. We simulate and debug the algorithm function, and add hardware constraints and timing simulation to the simulation platform. Finally, the simulation platform generates and loads onboard bitstream that is deployed to Zynq UltraScale+ MPSoC ZCU102. By analyzing the deployment experimental results of this framework, the correctness verification and performance evaluation are verified. We set its working frequency as 230 MHz for all designs and use a 16-bit fixed data type.

As shown in Table 1, the resources available on ZCU102 evaluation suite include 274 080 look-up tables (LUT), 14 400 distributed LUT random-access memory (LUTRAM) and 548 160 Flip flops (FF). 912 Block RAM (BRAM), each with a storage capacity of 36 KB, one high-speed embedded DDR4 SODIMN and 4 GB DDR capacity.

For this platform, processors include processing system (PS) and programmable logic (PL). PS is responsible for the preparation task before the reasoning and process scheduling task, PL is the core content, responsible for computing tasks; After acquiring network model weight parameters and input feature map data, PS generates flow control instructions according to the network model weight parameters and input feature map data storage information and sends the flow control instructions to PL. After receiving process control instructions, PL reads data from PS to calculate and finally feeds back the calculation results to the PS. After the PS sends a flow control command to the PL, the PS processes the nonlinear function until the PL writes result of the packet calculation into the DDR.

#### 4.2. Experimental results

We now present our experiments and then evaluate two parts: (i) Energy consumption, we optimize the code through AKGF and deploy the code to CPU-FPGA to test the energy consumption required to run the model through Vivado. (ii) Time-consuming, compare the running time of the codes of different models generated by AKGF.

<table>
<thead>
<tr>
<th>Platform</th>
<th>LUT</th>
<th>LUTRAM</th>
<th>FF</th>
<th>BRAM</th>
<th>BFUG</th>
<th>DDR</th>
</tr>
</thead>
<tbody>
<tr>
<td>ZCU102</td>
<td>274080</td>
<td>144000</td>
<td>548160</td>
<td>912</td>
<td>404</td>
<td>1</td>
</tr>
</tbody>
</table>

Table 1. Available resources on ZCU102.

#### Table 2. Performance comparison with other acceleration architectures based on GEMM ($512 \times 512 \times 16$).

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Frequency (MHz)</td>
<td>230</td>
<td>230</td>
<td>230</td>
<td>230</td>
</tr>
<tr>
<td>Performance efficiency (GOPS/W)</td>
<td>11.63</td>
<td>22.42</td>
<td>21.88</td>
<td><strong>25.81</strong></td>
</tr>
<tr>
<td>Time cost (ms)</td>
<td>2.431</td>
<td>1.278</td>
<td>1.491</td>
<td><strong>0.859</strong></td>
</tr>
</tbody>
</table>

The significance of bold values for best performance of the part.

Table 3. Performance comparison with other acceleration architectures based on GEMM ($2088 \times 2048 \times 16$).

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Frequency (MHz)</td>
<td>230</td>
<td>230</td>
<td>230</td>
<td>230</td>
</tr>
<tr>
<td>Performance efficiency (GOPS/W)</td>
<td>13.94</td>
<td>23.51</td>
<td>23.76</td>
<td><strong>26.32</strong></td>
</tr>
<tr>
<td>Time cost (ms)</td>
<td>28.183</td>
<td>18.084</td>
<td>21.856</td>
<td><strong>12.672</strong></td>
</tr>
</tbody>
</table>

The significance of bold values for best performance of the part.

In this section, our evaluation experiments for code generation are conducted on an Ubuntu 20.04 server with an Intel i9 9900k CPU and 32GB of DDR4 memory.

Experiments give the data size and type of model input as well as each DNN and overall evaluation results, including speedup, energy efficiency gain and throughput. To demonstrate the performance of AKGF, we select various types of applications in the field of image processing, including GEMM, AlexNet, VGGNet and YOLO.

#### 4.2.1. Case study for GEMM

For the GEMM experiment, the optimization goal of AKGF is the matrix multiplication of $2088 \times 2048 \times 16$ and $512 \times 512 \times 16$. The optimization method of matrix operation code is applied. First, a matrix operation operator is defined. The parameters of this operator include the input matrix ($A$, $B$) and the output matrix ($C$). At the same time, attributes such as unroll size are added to this operator. Among them ($MC$, $KC$, $MR$, $NR$) belong to the tile size, and $Kc$ belongs to unroll size. The selection of ($MC$, $KC$) is to enable the A matrix block of size $MC \times KC$ to be multiplexed in the L2 cache, and the selection of ($Kc$, $NR$) is to enable the B matrix block of size $Kc \times NR$ to be reused in the L1 cache, ($MR$, $NR$) is selected so that the output matrix block of size $MR \times NR$ can be reused in the CPU Register. These values are calculated or tuned according to hardware, and an empirical value is taken in the test.

#### 4.2.2. Case study for AlexNet

AlexNet consists of three pooling layers, two local response normalization (LRN) layers, five convolutional layers with integrated activation functions and three fully connected (FC) layers at the end. We sample the last three FC layers. Although the FC layer
Table 4. Performance comparison with other acceleration architectures based on AlexNet.

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Platform</td>
<td>ZCU102</td>
<td>ZCU102</td>
<td>ZCU102</td>
</tr>
<tr>
<td></td>
<td>XCZU9EG</td>
<td>XCZU9EG</td>
<td>XCZU9EG</td>
</tr>
<tr>
<td>Frequency (MHz)</td>
<td>230</td>
<td>230</td>
<td>230</td>
</tr>
<tr>
<td>DSP used (pcs)</td>
<td>844</td>
<td>723</td>
<td>673</td>
</tr>
<tr>
<td>Throughput (GOPS)</td>
<td>154.25</td>
<td>123.17</td>
<td>260.84</td>
</tr>
<tr>
<td>Performance efficiency (GOPS/W)</td>
<td>24.32</td>
<td>54.18</td>
<td>82.25</td>
</tr>
<tr>
<td>Time cost (ms)</td>
<td>21.349</td>
<td>13.721</td>
<td>14.973</td>
</tr>
</tbody>
</table>

The significance of bold values for best performance of the part.

Table 5. Performance comparison with other acceleration architectures based on VGG19.

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Platform</td>
<td>ZCU102</td>
<td>ZCU102</td>
<td>ZCU102</td>
</tr>
<tr>
<td></td>
<td>XCZU9EG</td>
<td>XCZU9EG</td>
<td>XCZU9EG</td>
</tr>
<tr>
<td>Frequency (MHz)</td>
<td>230</td>
<td>230</td>
<td>230</td>
</tr>
<tr>
<td>DSP used (pcs)</td>
<td>1203</td>
<td>824</td>
<td>727</td>
</tr>
<tr>
<td>Throughput (GOPS)</td>
<td>193.42</td>
<td>227.55</td>
<td>560.83</td>
</tr>
<tr>
<td>Performance efficiency (GOPS/W)</td>
<td>59.72</td>
<td>84.42</td>
<td>104.88</td>
</tr>
<tr>
<td>Time cost (ms)</td>
<td>51.946</td>
<td>37.573</td>
<td>36.932</td>
</tr>
</tbody>
</table>

The significance of bold values for best performance of the part.

we use a very small feature map, this can highlight the matrix computation advantage of the FPGA.

AKGF first expands DNN into the form of a computation graph and labels the operator nodes of each layer in the computation graph. A certain proportion (initial value 10%) of the conventional nodes are extracted for dynamic combination based on the cost model, and the operators that need to run on ARM and FPGA are simulated. The ARM side has ReLU, LRN, FC and other operators, which are further divided into multiplication and addition functions, and the ARM runtime is called to support the corresponding nonlinear calculation flow. Based on the optimization principle of polyhedron model, the operator of FPGA side further optimizes the code of digital logic side.

We are able to tag all the layers, form a group, and deliver them to the on-chip CPU core. The convolution kernels in each layer are implemented in the FPGA, and the activation functions in the layers are implemented using the CPU. Compared to other accelerators, our strategy achieves a corresponding speedup due to our efficient utilization of hardware computing resources.

4.2.3. Case study for VGG

For VGG19, we compare AKGF with state-of-the-art code optimization architectures. VGG19 consists of 16 convolutional layers, 3 fully connected layers, 3 max-pooling layers and a softmax layer. HeteroHalide chooses to fuse the first five convolutional layers and two pooling layers because the feature map transfer in these layers is heavy. For a fair comparison, we also fuse these seven layers. We use the same data and experimental platform to compare the performance of different architectures. Table 5 shows the latency and power efficiency comparisons.

Table 6. Experimental data for running 1000 image inferences based on YOLOv5.

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>No. 1</td>
<td>ZCU102</td>
<td>ZCU102</td>
<td>ZCU102</td>
</tr>
<tr>
<td></td>
<td>640×512</td>
<td>219.12</td>
<td>9.264</td>
</tr>
<tr>
<td></td>
<td>Time cost (ms)</td>
<td>82.25</td>
<td>166.259</td>
</tr>
</tbody>
</table>

Table 7. Performance comparison with other acceleration architectures based on YOLOv5.

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>No. 1</td>
<td>ZCU102</td>
<td>ZCU102</td>
<td>ZCU102</td>
</tr>
<tr>
<td></td>
<td>269.634</td>
<td>104.88</td>
<td>129.67</td>
</tr>
<tr>
<td>No. 2</td>
<td>71.52</td>
<td>94.42</td>
<td>129.67</td>
</tr>
</tbody>
</table>

4.2.4. Case study for YOLO

For YOLOv5, we test the performance of AKGF from different image sizes, the number of detected target types and different hardware models. In addition to ZCU102, we also choose VCK190 as our test board. The amount of data post-processed by YOLOv5 is related to the following parameters: image size (hw), the number of anchor ratios (ah), and the number of target types detected by the network (cn). The specific formula is: hw × ah × cn 

The significance of bold values for best performance of the part.

4.3. Limitations of using the AKGF

This section focuses on the experimental validation and discussion of the limitations of AKGF. We tried different types of CPU-FPGA heterogeneous platforms, including Xilinx ZYNQ-7000 and Intel Terasic De5-net FPGA.

In our experimental study, we attempted to incorporate references and comparisons to recent works, while also evaluating the advantages in terms of time consumption and performance. We analyzed and compared the work proposed by Almomany et al. on the power optimization of edge detection operators on Intel FPGA[24]. The article mentioned power consumption, stating that the average dynamic power consumption when using the Terasic De5-net FPGA is 7.687 W, based on their experiments. In our manuscript, we also performed code generation and logic design for the Sobel edge detection operator. However, the referenced article employed Intel’s FPGA board. To ensure fairness, we manually optimized the code for Terasic DE5-Net FPGA Development Kit, and our optimization method reduced
the dynamic power consumption to 6.211 W. This experimental result proves that on other types of FPGAs, our method is superior to the prior art in the actual measurement of average power consumption.

We also conducted an experimental comparison with the ZYNQ board classification model optimization method proposed by Wang et al. [25]. This article addresses the significant matrix computation issues in the AlexNet network and optimizes matrix operations using the OPENBLAS acceleration algorithm. It proposes a research and implementation of an embedded image classification method based on ZYNQ. Similar to our manuscript, the article delegates tasks to heterogeneous cores and employs FPGA for real-time image acquisition. The acquired images serve as input to the convolutional neural network model, which is implemented on the ARM side using the AlexNet network for image classification. Finally, the neural network model is deployed on the ZYNQ-7000 platform. The mentioned article reports a total time consumption of approximately 4.5 seconds for image classification. To ensure fair comparison in our experiments, we also used the ZYNQ-7000 platform and obtained a classification time of 1.2 seconds using the same dataset, achieving an optimization of nearly four times.

Combined with the experimental results, we summarized the limitations of AKGF use. AKGF is a DNN automatic deployment optimizer for CPU-FPGA heterogeneous platforms and has been tested in detail in this paper for the ZCU102. However, this optimizer only works for Xilinx heterogeneous FPGAs automatically, because it needs to use HLS to generate optimized C code, so Codegen is also one of the limitations of using AKGF to match the manufacturer’s code generation tool. AKGF targets different kinds of DNNs, and AKGF is effective at this stage for models with nonlinear functions as well as convolutional layers.

We implemented automatic code optimization and deployment of some DNNs for Xilinx ZYNQ FPGAs. In order to verify the dependency of the hardware architecture to our method, we also conducted manual tests against Intel FPGAs. For Terasic DeS-Net FPGA board, we follow the AKGF method, use OpenCL to manually optimize the code, and realize the optimization of power consumption and time on AlexNet model. We collected accurate values of the average power consumption and found that our optimization method still works.

5. CONCLUSIONS

In this work, we proposed a code generation deployment framework for accelerating optimized DNN for CPU-FPGA heterogeneous computing platforms. We first designed a TVM-based accelerator named AKGF, which could generate intermediate representation (IR) of neural network models based on TVM’s Halide IR. AKGF lowered the Halide IR to polyhedral IR. By using the polyhedral model to optimize the systolic array code, AKGF could process the code of DNN by heterogeneous kernel computing. DNN could be formed into the optimized data stream as well as giving full play to the computing performance of FPGA and CPU, realizing high performance and low latency deployment of CPU-FPGA. Test and verification were completed on DNN such as GEMM, AlexNet, VGG19 and YOLOv5s on heterogeneous CPU-FPGA boards, and the energy consumption and time consumption demonstrated an obvious optimization. Compared with the most advanced accelerator, the maximum speedup ratio reached 6.7, and the power consumption was optimized by two times.

ETHICAL APPROVAL

Written informed consent was obtained from all the participants prior to the enrollment (or for the publication) of this study.

AUTHORS’ CONTRIBUTION

Dong Dong participated in the design of the study and performed the statistical analysis. Dong Dong and Hongxu Jiang conceived of the study, and participated in its design and coordination and helped to draft the manuscript. Boyu Diao and Dong Dong prepared figures and carried out experiments. All authors read and approved the final manuscript.

FUNDING

This work was supported by the National Key Research and Development Program of China (No. 2021ZD0110202).

DATA AVAILABILITY

The datasets generated or analyzed during this study are available from the corresponding author on reasonable request.

REFERENCES


