PyAIE: A Python-based Compiler Framework for Versal ACAP Platforms

Hongzheng Tian, Shining Yang, Yoonha Cha, Sitao Huang
University of California, Irvine, USA
{hongzhet,shininy,yoonha.cha,sitaoh}@uci.edu

ABSTRACT

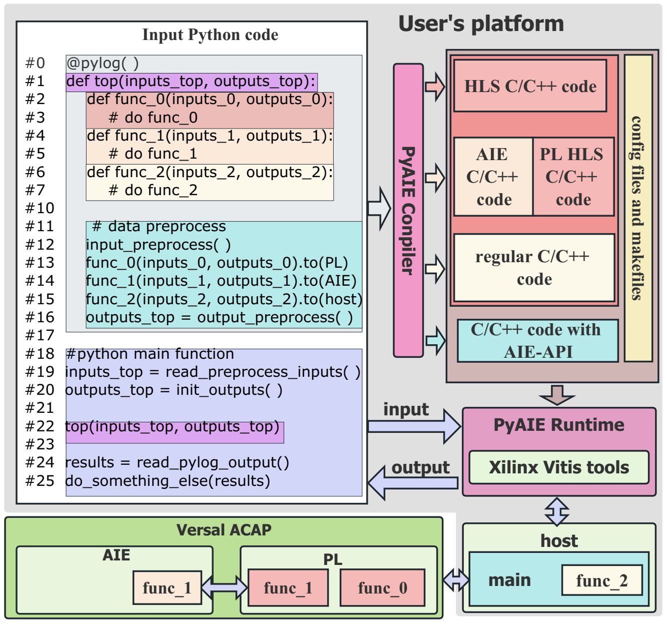
In order to satisfy the exploding demand from various computation-intensive applications, AMD-Xilinx released Versal, a heterogeneous Adaptive Compute Acceleration Platform (ACAP) with powerful heterogeneous acceleration capabilities. Most AI/ML applications use Python as their primary programming language these days; yet, AI engine codes as well as numerous Versal ACAP-based acceleration projects must be written in C/C++ with low-level intrinsics in order to take advantage of the Versal ACAP platform. To fill the gap of programming abstractions of applications and Versal ACAP platforms, we propose PyAIE, a Python-based compiler framework specifically targeting Versal ACAP architecture. PyAIE allows users to focus on algorithm-level designs without knowledge of the underlying low-level details. PyAIE automatically translates Python code into the optimized C/C++ code for AI Engines, programmable logic (PL), and host CPU, along with configuration script files, thereby completing the entire AI Engine-based system design. To the best of our knowledge, this is the first Python-based programming and compilation flow designed specifically for Versal ACAP platforms.

1 INTRODUCTION

In the conventional Versal ACAP design flow, full knowledge of the functionalities of the heterogeneous Versal platform is required. Application code, AI Engine (AIE), programmable logic (PL) code, host code, and configuration files all need to be written by developers. Manual development and maintenance of these codes for different level of architecture without automated tools is tedious and results in a long and error-prone development cycle. In order to overcome this inconvenience, we propose PyAIE, a compiler framework that allows users to efficiently develop the Versal ACAP based systems with Python. PyAIE automates the development process, reduces users' burden, and increases productivity by eliminating the need to understand the low-level hardware details. Users would design algorithms in Python instead of C/C++, using PyAIE intrinsic APIs. PyAIE then divides the user-written tasks into subtasks and maps them onto AIE tiles, PL, and the host platform, generating code automatically.

The main contributions of this work can be summarized as follows.

(1) To the best of our knowledge, our proposed flow is the first compilation flow to generate AIE kernel code, PL code, and host code from user's Python programs.
(2) PyAIE compilation flow enables vectorization optimization by generating AIE built-in vector registers and instructions.
(3) We demonstrate a prototype of Python-based system compiling and tuning tool that optimizes computation and data movement on heterogeneous Versal ACAP platforms.



```
#0  @pylog( )
#1  def top(inputs_top, outputs_top):
#2    def func_0(inputs_0, outputs_0):
#3      # do func_0
#4    def func_1(inputs_1, outputs_1):
#5      # do func_1
#6    def func_2(inputs_2, outputs_2):
#7      # do func_2
#10
#11   # data preprocess
#12   input_preprocess( )
#13   func_0(inputs_0, outputs_0).to(PL)
#14   func_1(inputs_1, outputs_1).to(AIE)
#15   func_2(inputs_2, outputs_2).to(host)
#16   outputs_top = output_preprocess( )
#17
#18 #python main function
#19 inputs_top = read_preprocess_inputs( )
#20 outputs_top = init_outputs( )
#21
#22 top(inputs_top, outputs_top)
#23
#24 results = read_pylog_output()
#25 do_something_else(results)
```

Figure 1: PyAIE Framework Overview.

2 RELATED WORKS

PyLog [3] is an algorithm-centric Python-based FPGA programming and synthesis flow, which was an inspiration for PyAIE. Users insert decorators (`@pylog`) into parts of the Python code, passing the decorated function into the PyLog process, which produces high-level synthesis (HLS) C code as an output. PyAIE extends PyLog and generates complete system design for Versal ACAP platforms. Section 3 discusses the details of PyAIE design.

Since AMD-Xilinx's launch of the Versal ACAP, researchers have published numerous accelerator and optimizer designs based on Versal platform [1, 2, 4]; yet, these designs still require significant amount of manual effort from programmers. Recently, a new work [5] proposed an innovative matrix multiplication accelerator design using a template-based approach. It has a high utilization of the Versal ACAP architecture. We incorporate this matrix multiplication accelerator into our flow and use it to implement generic tensor dot product operations. PyAIE specifically aims to improve the programmability and efficiency of Versal ACAP by providing a higher level of abstraction in the design flow and introducing an automated design process.

3 PYAIE DESIGN

Figure 1 is an overview of PyAIE framework. The left-hand side of the figure is an example of Python code input for the flow. Line 19 to 25 is the main function of this code. First, line 19 prepares the inputs, which, e.g., includes specifying the data type and size, and storing the data in files for the host code to read. Line 20 initializes the data type and size of the output. Then the function `top` in line 22 is decorated with the Python decorator `@pylog` (line 0 and line 1), which passes the function as well as the arguments as an input to the PyAIE compiler. Users can pass some manual instructions to the decorator in the form of arguments to achieve manual adjustments to PyAIE. Finally, in line 24, the Python main function reads the output files as the results of the function `top`, so that it can be used in the following portion.

Inside the `top` function, there are three sub-functions, `func_0`, `func_1` and `func_2` defined in line 2, line 4, and line 6 respectively. Line 12 does some preprocessing to the input data of the top function, and gets the inputs and outputs information for the three sub-functions. Line 13 to line 15 call those three sub-functions and indicate their targets by users. In this example, `func_0` is assigned to PL (using "to" function), meaning that PyAIE will use PL to implement this function. Similarly, `func_1` targets AIE. PyAIE will generate both AIE kernel code, as well as PL code to implement the data movers of AIE kernels to transfer the data between the AIE and host. Besides, if `func_1` is doing matrix multiplication, PL kernels are also needed to implement the helper functions to re-arrange the results from AIE kernels [5]. Lastly, `func_2` will target the host only, which will be implemented as a sub-function of the host source code.

At high level, PyAIE works in the following steps. The input Python code is passed to the PyAIE front-end analyzer, which analyzes the `top` function, gets the architecture of the code, and the data flow between sub-functions. For each sub-function, PyAIE passes the source code and the arguments to the PyLog front end, and PyLog intermediate representation (PLIR) is generated, which contains the information of the original source code, as well as the Versal ACAP programming information. In the next step, the PyAIE compiler takes over, analyzes the PLIR, and then generates kernels targeting different portions of Versal ACAP. The rest of `top` function will be only passed to the PyAIE compiler to generate the top host source. In addition, the configuration files and all the Makefiles will be generated according to the data flow information.

After PyAIE generates all the files needed, PyAIE Runtime will take over and call the AMD-Xilinx Vitis tools to compile the whole design. Users can also choose to do this manually. The results are stored in files and can be processed by the Python main function. If the `top` function is called again, then PyAIE Runtime will used again to execute the previously generated binary files.

4 OPTIMIZATIONS

In PyAIE, the computation of one-dimensional vector data in Python `numpy` is implemented with AIE matrix multiply APIs. For example, both `numpy.multiply` and `*` operations perform element-wise multiplication when the operands are one-dimensional vectors. In the generated AIE code, AIE API `aie::mmul` is called to perform the vectorized computation for `numpy.multiply`, while for `*` operation, we generate for loops to complete the element-wise multiplication.

As for the optimization of the data-intensive operations such as matrix multiplication (`numpy.dot`, `numpy.matmul`), we integrated

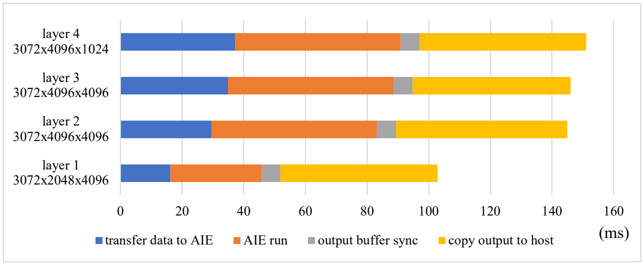


Figure 2: Execution Time Breakdown of Layers in MLP.

CHARM architecture[5] inside our framework. Upon encountering a large matrix multiplication, PyAIE will call the code generator of CHARM to generate the corresponding PL code and AIE code. For a 4-layer MLP application with matrix sizes of $3072 \times 2048 \times 4096$, $3072 \times 4096 \times 4096$, $3072 \times 4096 \times 4096$, and $3072 \times 4096 \times 1024$, the total execution time of AIE kernels is 190ms, which is $8,151\times$ faster than Python `numpy.dot` operation. We list the execution time breakdowns of each layer in Figure 2.

5 FUTURE WORKS

This is an on-going project that is being actively developed. Our future plan for PyAIE project can be summarized as follows.

(1) Further optimize generic data-intensive tensor operations in Python programs by improving the collaboration between AI Engine and programmable logic.
(2) Further explore various communication options provided by AIE, e.g., local memory, stream switch, and cascade channels. This can better increase the parallelism of tasks and increase throughput by using methods such as pipelines.
(3) Optimize the system generation in the current PyAIE flow. By fully leveraging Python programming features, we would like to enable more flexible mapping from Python functions to heterogeneous computing elements inside Versal devices.
(4) Improve resource allocation algorithms, especially in the cases where multiple functions share the AIE resource. A suitable method needs to be found to determine the number of AIEs allocated to each function.
(5) Closely integrate PyLog and PyAIE so that we can perform PL-AIE co-design and generate optimal PL accelerators and data movers for the corresponding AI Engine programs.
(6) Include more AIE APIs into PLIR to support wider range of Python functions on AIE.

In summary, PyAIE is a Python-based programming flow that can generate system design for Versal ACAP platforms. The goal of the project is to improve the programming efficiency of Versal platforms by leveraging the expressiveness of Python language, the capabilities of compiler analysis and optimization, and the heterogeneity of Versal ACAP platforms. We plan to further improve the optimization passes inside PyAIE to generate better kernel-level and system-level designs.

REFERENCES

[1] Ahmad Al-Zoubi, Gianluca Martino, Fin H. Bahnsen, Jun Zhu, Holger Schlarb, and Goerschwin Fey. 2022. CNN Implementation and Analysis on Xilinx Versal ACAP

at European XFEL. In *2022 IEEE 35th International System-on-Chip Conference (SOCC)*. IEEE, Belfast, United Kingdom, 1–6. https://doi.org/10.1109/SOCC56010.2022.9908101

[2] Prasanth Chatarasi, Stephen Neuendorffer, Samuel Bayliss, Kees Vissers, and Vivek Sarkar. 2020. Vyasa: A High-Performance Vectorizing Compiler for Tensor Convolutions on the Xilinx AI Engine. In *2020 IEEE High Performance Extreme Computing Conference (HPEC)*. IEEE, Waltham, MA, USA, 1–10. https://doi.org/10.1109/HPEC43674.2020.9286183

[3] Sitao Huang, Kun Wu, Hyunmin Jeong, Chengyue Wang, Deming Chen, and Wen-Mei Hwu. 2021. PyLog: An Algorithm-Centric Python-Based FPGA Programming and Synthesis Flow. *IEEE Trans. Comput.* 70, 12 (2021), 2015–2028. https://doi.org/10.1109/TC.2021.3123465

[4] Chengming Zhang, Tong Geng, Anqi Guo, Jiannan Tian, Martin Herbordt, Ang Li, and Dingwen Tao. 2022. H-GCN: A Graph Convolutional Network Accelerator on Versal ACAP Architecture. https://doi.org/10.48550/ARXIV.2206.13734

[5] Jinming Zhuang, Jason Lau, Hanchen Ye, Zhuoping Yang, Yubo Du, Jack Lo, Kristof Denolf, Stephen Neuendorffer, Alex Jones, Jingtong Hu, Deming Chen, Jason Cong, and Peipei Zhou. 2023. CHARM: Composing Heterogeneous AcceleRators for Matrix Multiply on Versal ACAP Architecture. In *The 2023 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (FPGA '23, 12)*. Association for Computing Machinery, New York, NY, USA, 1–12. https://doi.org/10.1145/3543622.3573210