Pythonプログラムからの高位合成による演算器選択法の一検討

島田　征弥[†1]　久我　守弘[†2,a)]

概要：FPGA は再構成可能な特性により，特定のタスクに最適なハードウェアアクセラレーションを実現できる利点があるが，プログラムや最適化が困難である．これに対し，高位合成（HLS）を用いることでプログラムの簡略化が可能だが，異なる言語や知識が必要であり，複雑さが増す．先行研究では，pylog が手順を簡素化し，全体を自動化できるが，実行ファイルの再作成やストリーミング処理に対応していない課題がある．本研究では，解析やコンパイル段階で情報を抽出し，適切な実行ファイルを選択することで再作成を省略し，PyLog にストリーミング処理を対応させる手法を検討している．

1. はじめに

近年，アプリケーションはますます複雑化し，それに伴い高い計算能力や処理能力が求められるようになり，従来のソフトウェア開発モデルだけでは十分なパフォーマンスを発揮できない場面が増えている．

このような要件を満たすために，設計者は FPGA（Field-Programmable Gate Array），などのようなアクセラレータを導入し，特定のアプリケーションに対応した組み込みシステムを利用している．この中で FPGA はその再構成可能な特性により，特定のタスクに最適なハードウェアアクセラレーションを実現できるという利点がある．

CPU-FPGA システムは，組み込みシステムの一般的なプラットフォームの一つであり，CPU の汎用性と FPGA のアクセラレーション性能を兼ね備えている．しかし，効果的かつ効率的にプログラムし，最適化することはソフトウェアを扱う人にとって困難であるという問題がある．一般的な FPGA のプログラミング開発フローは，まず Verilog や VHDL などのハードウェア記述言語（HDL）でレジスタ転送レベル（RTL）の FPGA プログラミングを行う．その後，FPGA 合成ツールを使用して RTL デザインを FPGA ビットストリームに合成する．

高位合成（HLS）という C/C++/OpenCL[1〜4] などの抽象度の高い言語から直接ハードウェアを合成する技術を用いることは可能であるが，適切な HLS プラグマや，ソースコードを書き換える必要がある．そのため，アプリケーションによっては，ホスト側で python ベースのライブラリやツールの利用，FPGA 側では C ベースの高位合成や Verilog ベースの設計に加えてインターフェースの設定が必要になる．これらは全く異なる知識を必要とし，実装においてそれぞれ設計上で考慮する必要があり，この複雑さが CPU-FPGA 組み込みシステムのプログラミングを困難にしている．

Python や Scala，Haskell などの高水準言語をハードウェア記述言語（HDL）として使用する研究もいくつか存在する．[5〜8] 先行研究で提案されている pylog[9] では，これらの手順を簡素化し，全体を自動で実行することが可能であるため，ハードウェアの知識を必要とせず，アルゴリズムと計算フローに集中することができる．

しかし，pylog の実行する関数を少し変えるだけでも実行ファイルを作り直す必要があり，再びすべての手順を行う必要がある．また，多くのライブラリに対応しているが streaming 処理に対応していない．

そこで本研究では，解析やコンパイルの段階で必要な情報を抽出し，適切な実行ファイルを選択することで再び実行ファイルを生成させずに実行させる手法，また PyLog にストリーミング処理を対応させることを検討する．

2. PyLog

PyLog は Python ベースの FPGA 向け高レベルプログラミングフローであり，システム設計，機能シミュレーション，柔軟な設計空間の探索が可能になる．

2.1 PyLog の処理フロー

図 1 に，FPGA デザインスタック全体における PyLog の

[†1] 現在，熊本大学 大学院　自然科学教育部
[†2] 現在，熊本大学 熊本大学大学院先端科学研究部
[a)] kuga@cs.kumamoto-u.ac.jp

処理のフローを示す．図は先行研究 [9] より引用している．

PyLog コンパイラは，Python コードを入力として受け取り，HLS のプラグマを使用して最適化され，HLS で合成可能な C コードにコンパイルする．Python 言語を使用することで，開発者が実装の詳細をあまり考えずにアルゴリズムと計算フローの記述に集中できるようにする．これにより，PyLog コンパイラは，可能な実装と最適化を検討する従来の FPGA 開発者の負担を軽減するように設計されている．

2.2 PyLog プログラムの例

図 2 に，ホストとアクセラレータの両方の動作を記述する PyLog プログラムの簡単な例を示す．この例には，preprocess と compute という 2 つのトップレベル関数が含まれている．compute 関数は Python デコレータ`@pylog`で宣言され，PyLog カーネル関数となり，PyLog によって FPGA 上のハードウェアアクセラレーターに合成される．`@pylog` デコレータを使用すると，プログラマは既存の Python コードでアクセラレータ関数を簡単に指定できる．例で示したように，ホストとアクセラレータは同じ抽象度レベルで Python を使ってプログラムされている．PyLog は，ホストプログラミングと FPGA アクセラレータプログラミングの抽象レベルのギャップを埋め，効率的なシステムレベルのホストとアクセラレータの設計を可能にすることができる．

2.3 PyLog の全体フローとシステムアーキテクチャ

図 3 に，PyLog の全体的なフローとターゲット FPGA システムのアーキテクチャを示す．PyLog プログラムを実行すると，`@pylog` デコレータが PyLog コンパイラを呼び出し，装飾された PyLog カーネル関数を HLS C コードに

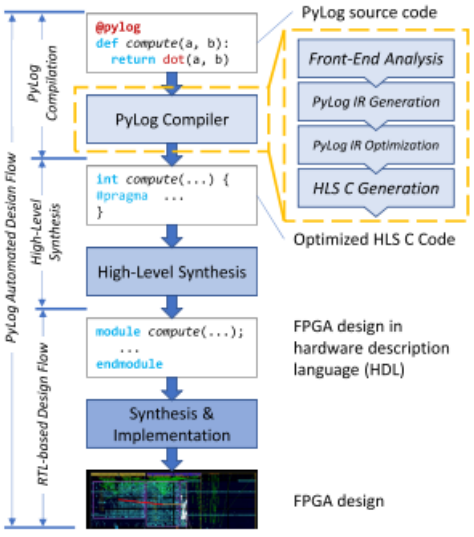

図 1 PyLog による FPGA 設計フロー
Fig. 1 FPGA design flow with PyLog

```
1  def preprocess(data):
2    ... # data pre-processing that runs on the host
3
4  @pylog
5  def compute(inputs): # top FPGA kernel function
6      def do_work(data): # some helper function
7          ...
8      for d in inputs:
9          do_work(d)
10   ...
11
12 inputs = preprocess(data) # data pre-processing
13 result = compute(inputs)  # synthesize/call FPGA
```

図 2 PyLog の簡単なコード例
Fig. 2 A simple example of a PyLog code

コンパイルする．その後，ハードウェアロジックを合成し，すべてのシステムコンポーネントを統合して完全な FPGA デザインを作成する．生成された FPGA ビットストリームは，FPGA 上のハードウェアリソースと相互接続の構成に使用される．PyLog プログラムの残りの部分は標準的な Python インタプリタによって解釈され，ホスト CPU 上で実行されるホストコードとなる．装飾された関数が呼び出されると，PyLog ランタイムが起動され，FPGA のプログラミング，メモリの割り当て，計算を高速化するための FPGA の起動が行われる．PyLog ランタイムは，Xilinx 社の PYNQ[10] システムとランタイムライブラリである XRT が使用される．ターゲットとする実行環境は，図 3 に示すように PCIe のベースの FPGA プラットフォームだけでなく，Zynq プログラマブル SoC 上の Pynq システムで利用できる．PYNQ-Z2 は，Python Productivity for Zynq（PYNQ）プラットフォームの一部であり，Xilinx 社が提供する低コストなハードウェアプラットフォームである．このプラットフォームは，Python プログラミング言語と統合されており，Zynq SoC（System on Chip）ベースのプログラムを開発するためのオープンソースのフレームワークとして設計されている．以下に特長をまとめる．

(1) 拡張ポートやコネクタを備えており，さまざまな周辺デバイスやセンサーとの接続ができる
(2) 低コストで入手可能なハードウェアプラットフォーム
(3) Python プログラミング言語と Jupyter Notebook を使用してプログラムを記述および実行することができ，ハードウェアの制御やデータ処理が容易に可能である

PyLog コンパイラは Python ライブラリとして提供されており，FPGA 合成や FPGA アクセラレータを実行するには，標準の Python インタプリタでプログラム全体を実行するだけで行うことが可能である．PyLog は装飾された計算関数をコンパイルし，FPGA IP を生成し，他の IP と統合して完全な FPGA 設計にし，最後に FPGA ハードウェア設計を合成して FPGA ビットストリームと構成ファイルを取得する．基礎となる CPU と FPGA の相互作用は

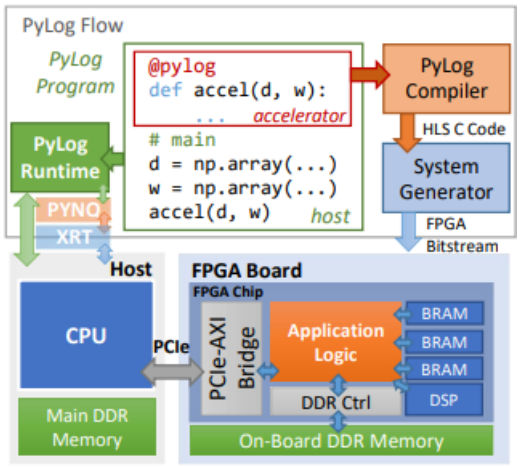

図 3 PyLog の全体フローと FPGA のシステムアーキテクチャ
Fig. 3 Overall flow of PyLog and FPGA system architecture

すべて PyLog ランタイムによって処理される．

2.4 PyLog のモード

PyLog を使用するには，pylog をインポートし，FPGA アクセラレータに合成したい関数に PyLog デコレータ@pylog を追加する．以下の図 4 に例を示す．さらに PyLog は，`@pylog` デコレータに “mode” 文字列を渡して PyLog のモードを設定することができる．利用可能な PyLog モードの一覧を以下に示す．

(1) cgen：デコレータされた関数を解析後に中間言語に変換し，最適化された HLS C コードを生成する．
(2) hwgen：cgen によって最適化された HLS C コードに高位合成を行い，FPGA 合成を実行する．bit ファイルや hwh ファイルなどの実行ファイルが生成される．
(3) deploy：生成された実行ファイルのプログラムを実行し，FPGA をコールして結果を収集する．
(4) pysim：PyLog でデコレータされたコードを Python インタプリタでシミュレートすることが可能．

また，@pylog デコレータに “board” 文字列を渡してターゲットの FPGA ボードを指定することができる．現在は pynq-z2，pynq-z1，zedboard，および ultra96 の各ボードをサポートしている．

2.5 PyLog のコンパイルと合成

PyLog のフローは，完全に自動化された FPGA プログラミングおよび合成フローであり，PyLog コンパイラ，PyLog システムジェネレーター，PyLog ランタイムの 3 つの部分で構成されている．PyLog コンパイラは，PyLog のソースコードを HLS ツールで合成可能な最適化された HLS C コードに変換する．現在サポートされている HLS ツールは，Xilinx Vivado HLS[11] と Merlin コンパイラ [12] である．

PyLog コンパイラのコンパイルは，以下の手順となる．

(1) フロントエンド解析と PyLog 中間表現（PLIR）の生成
(2) 型の推論とチェック，
(3) コードの最適化
(4) HLS C コードの生成

3. 提案手法

本章では解析の段階で必要な情報を抽出し，適切な実行ファイルを選択することで再び実行ファイルを生成させずに実行させる手法，また PyLog にストリーミング処理を対応させることを検討する．AXI Stream は，デバイス間でデータを連続的に送受信するためのインタフェースであり，データの受け渡しに関する約束事やプロトコルを定義しており，デバイス間のシームレスなデータ通信を実現する．このような仕組みを利用することで，異なるデバイス間でデータを効率的にやり取りすることが可能である．

3.1 提案フローの概要

図 5 に提案するフローの概要を示し，以下の流れで実装を考える．

(1) PyLog IR の出力前に必要な情報を抽出
(2) 情報を条件と照らし合わせ，マッチングした bit ファイルを実行
(3) マッチングする bit ファイルがなかった場合，最後の手順まで行い bit ファイルを生成する．

3.2 情報の抽出

PyLog の解析や型推論，型チェックを利用して，マッチ

```python
import numpy as np
from pylog import *

@pylog
def vecadd(a, b, c):
    for i in range(1024):
        c[i] = a[i] + b[i]
    return 0

if __name__ == "__main__":
    length = 1024
    a = np.random.rand(length).astype(np.float32)
    b = np.random.rand(length).astype(np.float32)
    c = np.random.rand(length).astype(np.float32)

    vecadd(a, b, c)
```

図 4 pylog デコレータの例
Fig. 4 Example of a pylog decorator

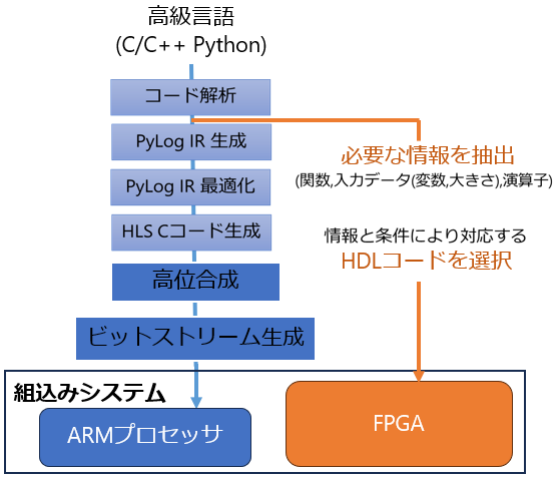

図 5 提案手法の概要フロー
Fig. 5 Overview flow of the proposed method

```
PYLOG_info [{'a': ('int64', (1024,)), 'b': ('int64', (1024,)), 'c': ('float64', (1024,))}, 'Mult']
PYLOG_info [{'a': ('int64', (1024,)), 'b': ('int64', (1024,)), 'c': ('float64', (1024,))}, 'range', 'Mult']
PYLOG_info [{'a': ('int64', (3, 3)), 'b': ('int64', (3, 3)), 'c': ('float64', (3, 3))}, 'dot']
```

図 6 抽出した PyLog info
Fig. 6 Extracted PyLog info

ングに必要な情報を抽出する．以下に実際に抽出した情報を図 6 に示す．

python の辞書機能を用いて入力の変数をキーとして，“arg inf” という変数から入力の変数の型とサイズを登録，さら “node” の情報から，演算子を加えそれらをまとめた “PYLOG info” として抽出している．入力の変数は初めから指定することもできるが，ここでは指定していないため PyLog の型推論により，自動的に推論されたものが抽出されている．例えば 1 行目の PYLOG info に注目すると，“a” と “b” という入力それぞれが “int64” の型を持ち “1024” のサイズの配列であり，行列乗算の演算を行う関数であるという情報を持っている．ここで演算子の情報は今回取り出した変数の関係上 “Mult” となっているが乗算を表す演算子 “ * ” の状態で取り出すことも可能である．

また，2 行目は行列演算に for 文が組み込まれているため，“range” という情報を抜き出しループ関数であると分かるようにしている．これは例えば 2 重ループ，3 重ループになると PYLOG info に含まれる “range” の数が 2 つ 3 つと増えることで判断できるようになっている．

そして，3 行目のように “dot” や “map” のような pylog が対応している高レベル演算子であればわざわざ中身を展開せずともそのまま情報を受け取ることができる．

3.3 マッチング

3.2 で抽出した情報からマッチングさせる手法について，以下の流れで処理を行う．概要を図 7 に示す．

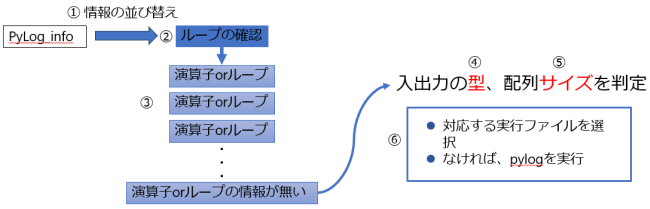

図 7 マッチングの概要
Fig. 7 Overview of matching

① 選択しやすいように情報を並び替える
② ループが存在するかどうかを判断する，存在しない場合④へ
③ 演算子の情報が終わるまでループか演算か判断しネストを進む
④ 入力や出力の配列の型を判定
⑤ 配列のサイズを判定
⑥ ここまでの判定結果に当てはまる bit ファイル（+hwh ファイル）が存在すれば実行，なければ pylog のハードウェア設計を実行し，この改装に保存する．

具体例として図 6 の 2 行目を実行する場合を考える．まず情報は得られる順番に配列や辞書に登録されているため，判断しやすいように条件文で参照する順番に並び替える．これは情報が得られる時間が速いものから PYLOG info に格納されているためである．そして条件文によりループが存在するかどうかを判断し，存在しなかった場合，演算子の特定（高レベル演算子）後，すぐに入出力の型，サイズの判定を行う．ループが存在する場合は，演算子もしくはループの情報がなくなるまでネストを進み，同じように判定していく．ここではループは存在するため，ループと判定し，再び判定，次は “Mult” を得て再び判定，そして演算子情報がなくなったので今度は入出力の型（ここでは int64 と float64 が一致するかどうか）を判定する．そしてサイズはこの例では 1024 であるため 1024 以上かどうか判定する．判定後，その情報に対応する実行ファイルが存在する場合（以前に一度 pylog によって実行ファイルが生成されていた），その実行ファイルの場所を値として返す．存在しなければそのまま pylog を実行し，bit ファイルと hwh ファイルの実行ファイルを生成する．

4. 評価

今回提案した手法の有効性を調べるために，各種項目について評価を行った．

（1）評価環境および評価指標

評価環境は表 1 に示す．評価指標としては，そのまま PyLog を実行した場合と，提案した手法を用いた場合における pylog の処理時間を比較することで提案手法の有効性を評価する．

表 1　評価環境
Table 1　evaluation environment

ターゲットボード	TUL 社製 PYNQ-Z2
ターゲットデバイス	AMD/Xilinx 社製 Zynq 7000, XC7Z020
ホスト計算機	Intel i7-6850K@3.60GHz, Ubuntu 22.04.3
高位合成	AMD/Xilinx Vitis HLS v2022.2
FPGA 設計ツール	AMD/Xilinx Vivado v2022.2

表 2　行列乗算の場合の実行時間
Table 2　Execution times in the case of matrix multiplication

行列サイズ	timeA[s]	timeB[ms]
96	470	6.21
1,024	469	5.78
4,096	593	5.81

表 3　内積計算の場合の実行時間
Table 3　Execution times in the case of inner product calculations

行列サイズ	timeA[s]	timeB[ms]
5×5	500	6.12
1024×1024	523	9.81
4096×4096	558	10.74

（2）結果および考察

今回対象とするのはループを含めた四則演算と高レベル演算（dot）とし，入力と出力はすべて配列であるとする．

（3）処理時間

PyLog によって python コードからハードウェア生成までの時間（timeA）と提案手法でハードウェア生成を省略できた場合の時間（timeB）の比較を行う．まず四則演算（ここでは行列乗算）の場合を表 2 に示す．

内積計算の場合を表 3 に示す．

それぞれの表からサイズによって pylog によるハードウェア設計の時間に差はあるが提案手法と比較すると全体的に大きく処理時間を減少させることが可能であると分かる．

（4）streaming 処理

ストリーミング処理に関しては配線の問題により実際にハードウェア設計まで自動で実装することはできなかったが，解析後中間言語から最適化によって HLS C を生成する際にプラグマを挿入し，ストリーム用の型に変換を行い，高位合成ツールに適用させることはできた．生成した HLS C コードの例を以下の図 8 に示す．

pylog デコレータのモードに “stream” というモードを追加すれば，明示的にストリーミング演算器を選択し実装することができるようにする．ストリーミング処理を行う場合，CPU を介さずに主記憶から実装する関数にデータを直接転送する DMA（Direct Memory Access）コントローラーを用いる必要がある．DMA を追加する場合，Vivado での自動配線の機能ではすべての配線を完了できないため，Vivado で用いる tcl スクリプトには実装する関数およびその入力変数名等を用いてに配線できるように考慮する必要がある．現在，本機能の実装については完了していないが，以下の図 9 を再現するように tcl スクリプトの設定を行うという実装の目途は立っている．

```
Generated C Code:
#include "ap_int.h"
#include "ap_fixed.h"
#include "hls_math.h"
#include "hls_stream.h"
#include "ap_axi_sdata.h"
typedef ap_axiu <32, 0, 0, 0> trans;
typedef ap_axiu <64, 0, 0, 0> tran;

int vec(hls::stream<trans>& a, hls::stream<trans>& b, hls::stream<tran>& c)
{
#pragma HLS INTERFACE axis port=a
#pragma HLS INTERFACE axis port=b
#pragma HLS INTERFACE axis port=c
#pragma HLS INTERFACE s_axilite port=return bundle=control

        for (int i = 0; i < 4096; i += 1){
                tran tmp;
                trans a_data, b_data;
                a.read(a_data);
                b.read(b_data);
                tmp.data = a_data.data * b_data.data;

                c.write(tmp);
        }
        return 0;
}
```

図 8　streaming 処理用に変換した HLS C コード
Fig. 8　HLS C code converted for streaming processing

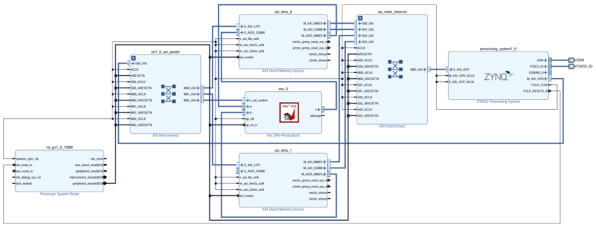

図 9　streaming 使用のデザイン例
Fig. 9　HLS C code converted for streaming processing

8 に示す．

5. まとめ

近年，アプリケーションが複雑化し，高い計算能力や処理能力が求められている．そのため，FPGA や GPU，ASIC などのアクセラレータを導入し，特定のアプリケーションに対応した組み込みシステムが利用されている．特に FPGA は再構成可能な特性により，特定のタスクに最適なハードウェアアクセラレーションを実現可能である．

CPU-FPGA システムは，CPU の汎用性と FPGA のアクセラレーション性能を兼ね備えているが，プログラムや最適化を行うことが困難である．高位合成を利用すると，高水準言語から直接ハードウェアを合成することが可能であるが，適切な設定が必要である．PyLog は手順を簡素化し，全体を自動で実行することができるが，関数を変更する際に再度手順を行う必要がある．

本研究では，再度実行ファイルを生成せずに実行できる

ようにする手法や PyLog にストリーミング処理を対応させる方法を検討した．PyLog の解析や型推論システムから情報を取り出し，演算子や型，サイズからマッチングを行い，適切な実行ファイルを選ぶことで可能な限りハードウェア設計の手順を省略することが可能であると分かった．またストリーミング処理を対応させるための変換を行い，streaming 処理用の HLS C コードへの自動変換は可能であるとわかった．

参考文献

[1] Xilinx, “Vivado high-level synthesis,” 2021. [Online]. Available: https://www.xilinx.com/products/design-tools/vivado/integration/esl-design.html
[2] Intel, “Intel high-level synthesis compiler,” 2021. [Online]. Available: https://www.intel.com/content/www/us/en/software/programmable/quartus-prime/hls-compiler.html
[3] Xilinx, “Xilinx SDAccel development environment,” 2019.[Online]. Available: https://www.xilinx.com/products/designtools/software-zone/sdaccel.html
[4] Intel, “Intel FPGA SDK for OpenCL software technology,” 2020.[Online]. Available: https://www.intel.com/content/www/us/en/software/programmable/sdk-for-opencl/overview.html
[5] Chisel,“Chisel/FIRRTL hardware compiler framework,” 2021.[Online]. Available: https://www.chisel-lang.org/
[6] Clash,“Clash: A modern, functional, hardware description language.” 2021. [Online], Available: https://clash-lang.org/
[7] PyMTL3,“PyMTL3 (Mamba), An open-source python-based hardware generation, simulation, and verification framework,” 2021. [Online]. Available: https://github.com/pymtl/pymtl3
[8] PyRTL, “PyRTL,” 2021. [Online]. Available: https://ucsbarchlab.github.io/PyRTL/
[9] PyLog: Sitao Huang , Kun Wu, Hyunmin Jeong, Chengyue Wang, Deming Chen, Fellow, IEEE, and Wen-Mei Hwu, Fellow, IEEE ”An Algorithm-Centric Python-Based FPGA Programming and Synthesis Flow” IEEE Transactions on Computers Volume:70, pp:2015-2028:Issue:12, 01 December 2021. [Online]. Available: https://github.com/hst10/pylog.git
[10] PYNQ. 2021. [Online]. http://www.pynq.io/
[11] Xilinx, “Vivado high-level synthesis,” 2021. [Online]. Available: https://www.xilinx.com/products/design-tools/vivado/
[12] Falcon computing, “Merlin compiler,” 2021. [Online]. Available: https://www.falconcomputing. com/ merlin-fpga-compiler/
[13] J. Zhao, L. Feng, S. Sinha, W. Zhang, Y. Liang, and B. He,“Performance modeling and directives optimization for high level synthesis on FPGA,” IEEE Trans. Comput.-Aided Des. Integr. Circuits Syst., vol.39, no.7, pp.1428–1441, Jul. 2020.