高位合成ツール Pylog におけるストリーミング処理への適用

川口颯太[1]　久我守弘[2]

概要：FPGA（Fieled Programable Gate Array）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．高位合成（High-Level Synthesis，HLS）を利用することで，C/C++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ことが可能であるが，多くの知識を必要としプログラミングを困難にしている．先行研究である“PyLog”は，FPGA のプログラミングフローを簡素化し，ハードウェア合成を自動で実行する．しかし，PyLog は FPGA での高速処理で有用なストリーミング処理をサポートしていない等の制限がある．そこで，行列乗算を例としてストリーミング処理に対応するための機能拡張を行った．実装手法として，行列サイズの抽出，ストリーミング処理に対応した HLS C コードの作成，ハードウェア生成時のテンプレートファイルの作成を行った．その結果，従来の PyLog を用いた実装と比較して実行時間の短縮が達成され，性能向上に寄与することを確認した．

キーワード：FPGA，Python，高位合成，ストリーミング処理

The Application of the High-Level Synthesis Tool PyLog to Streaming Processing

SOTA KAWAGUCHI[1]　MORIHIRO KUGA[2]

Abstract: Field-Programmable Gate Arrays (FPGA) have the advantage of enabling hardware acceleration optimized for specific tasks due to their reconfigurability. High-Level Synthesis (HLS) allows direct synthesis of hardware from high-level languages such as C/C++, making it possible to design hardware with less complexity. However, it requires considerable knowledge and can make programming difficult. The previous research “PyLog” simplifies the FPGA programming flow and automates hardware synthesis. However, PyLog has limitations, such as not supporting streaming processing, which is essential for high-speed FPGA processing. To address this, we extended PyLog to support streaming processing, using matrix multiplication as an example. The implementation involved extracting matrix sizes, creating HLS C code for streaming processing, and generating template files during hardware generation. As a result, we confirmed that execution time was reduced compared to traditional implementations using PyLog, contributing to improved performance.

Keywords: FPGA, Python, High-Level Synthesis, Streaming Processing.

1. はじめに

近年，IoT の普及に伴い，組込みシステムに求められる計算性能や消費電力効率，ならびに柔軟性が飛躍的に高まっている．そのような状況下で，CPU と FPGA を組み合わせたシステムは，汎用的な制御処理を CPU が担いつつ，特定処理を FPGA がハードウェアレベルで加速することにより，高い性能と柔軟な再構成機能を両立できるため注目を集めている．

一方で，FPGA にハードウェアロジックを実装するためには，従来の高級言語プログラミングとは異なる専門知識が要求される．具体的には RTL（Register Transfer Level）設計や論理合成・配置配線のプロセスを理解し，さらに動作の検証や最適化のために煩雑なデバッグ作業を行う必要がある．このように FPGA を活用するためにはハードウェア開発特有の知識が不可欠であり，その学習コストの高さが導入の障壁となっている．

一般的に FPGA のプログラミング開発フローは Verilog HDL や VHDL などのハードウェア記述言語を用いてレジスタ転送レベル（Resister Transfer Level）から開発を行う．

高位合成（High Level synthesis)という C/C++[1~3]などの抽象度の高い言語からハードウェア開発を行うことも可能であるが，最適化のために適切な pragama の挿入や用途に合わせたコーディングが必要である. そのため, CPU-FPGA システムの有用性が認識されつつも，実際の開発・導入には高いハードルが存在するのが現状である．

本研究では，こうした「FPGA は有用だが，利用の敷居が高い」という課題に対処するために，組込みシステム開発フローの簡素化とプログラミングの容易化，そして必要な最適化手法を検討・提案することで，より多くの開発者が CPU-FPGA システムの潜在的な性能を引き出せるようにすることを目的としている．

本研究では，先行研究で提案されている python ベースの FPGA プログラミングフレームワークである“PyLog”[4]を利用する．PyLog では高位合成や合成を行うための RTL レベルの記述を必要とせず，自動で実行することが可能であるためハードウェアの専門知識をあまり必要としない．しかし，対応している機能に制限があり，FPGA を扱う際に有用なストリーミング処理に対応していないといった問題点が挙げられる．そこで，本研究では PyLog をストリーミング処理に対応させ，PyLog の機能拡張，そして実行時間の短縮を図る．

1 熊本大学　大学院自然科学教育部
2 熊本大学　大学院先端科学研究部

2. PyLog

Pylog は Python プログラムをベースとした FPGA 向け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が可能である．一般的な Python 構文を使用して記述できるため比較的容易に設計を行うことができる．

2.1 PyLog の実行フロー

図 1 に Pylog を利用した FPGA 生成フロー全体を示す．図は先行研究[4]より引用している．

Pylog コンパイラでは，Python コードを入力として受け取り，pragma を使用して最適化された HLS C コードを出力する．Python コードを入力として利用するため，開発者は実装の詳細をあまり記述することなく，アルゴリズムと計算フローの記述に集中することができる．このコンパイラでは実装や最適化を考慮する従来の FPGA 開発者の負担を軽減することができる．この実装フローではアルゴリズムとハードウェア設計は可能な限り分離されている．

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が同じ抽象化レベルの Python プログラムを利用している例を示す．
この例では 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 関数では Python デコレータの@pylog で装飾される．その後 Pylog によって FPGA 上のハードウェアアクセラレータに合成される．Pylog はホストと FPGA アクセラレータ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協調設計を可能にすることができる．

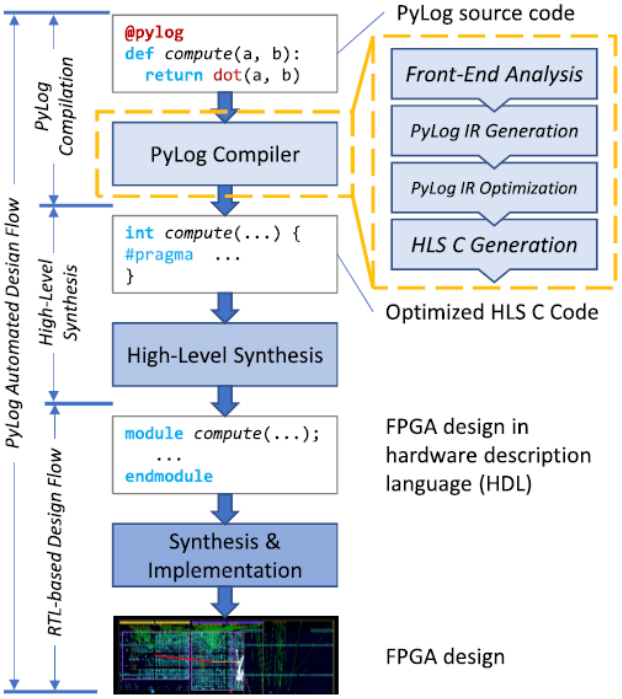

図 1 PyLog による FPGA 設計フロー[4]
Fig.1 FPGA design flow with PyLog. [4]

Listing 1. A Simple PyLog Example

```
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 のプログラムコード例
Fig.2 Example of a PyLog code.

2.3 PyLog のシステムアーキテクチャ生成フロー

図 3 のように 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，装飾された関数を HLS C コードを出力する．Pylog の残りの部分のプログラムはホスト CPU 上で実行されるコードである．装飾された関数が呼び出されると Pylog ランタイムが起動し，FPGA のプログラミング，計算の高速化のための FPGA 起動が行われる．Pylog ランタイムは AMD/Xilinx 社のランタイムライブラリである PYNQ[5]と XRT が利用されている．

PyLog コンパイラは Python ライブラリとして提供されている．FPGA 合成や FPGA アクセラレータを実行するには，ユーザは標準の 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．装飾された Python 関数の compute が呼び出されると（図 2 の 13 行目），PyLog コンパイラが呼び出され，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 IP を生成する．最後に FPGA ハードウェア設計を合成し，FPGA ビットストリーム（*.bit）と構成ファイル（*.hwh）を取得する．

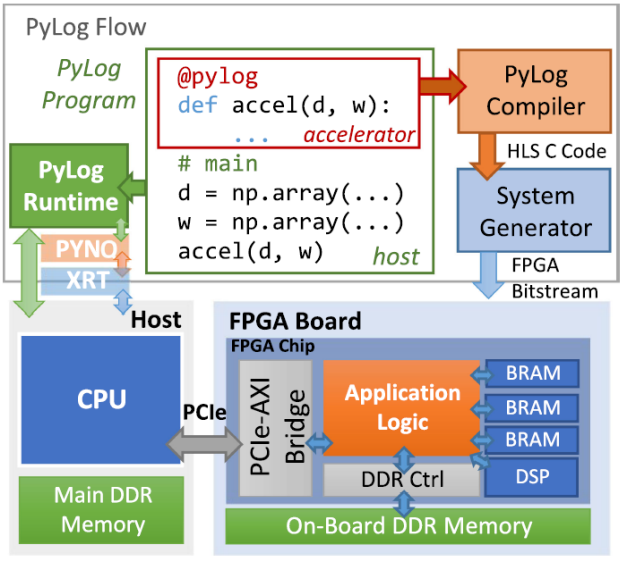

図 3 PyLog のシステムアーキテクチャ生成フロー
Fig.3 PyLog system architecture generation flow.

また，FPGA アクセラレータプログラムを実行するために，ユーザは FPGA 固有のコードを記述する必要はなく，デコレータ@pylog を除き，通常の Python コードと非常によく似た同じ記述を利用する．

3. ストリーミング処理

AXI4- 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，データ転送の制御と同期を容易にするための図 4 のような信号線のセットである．また，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．これにより，プロセッサ部―FPGA 部間でデータ転送を効率的に行うことが可能である．

AXI4-Stream では，マスタ側からスレーブへ clock に同期してデータの転送を行う．マスタがデータを送る際には，データが有効であること示す TVALID 信号と共にデータを TDATA として送信する．スレーブ側はクロックの立ち上がりでその際データを受け取る．この際，スレーブ側が TREADY 信号をアサートしていたときは，マスタは送信できたと解釈し次サイクルでは次のデータを送ることができる．一方で TREADY 信号がネゲートであった場合はスレーブ側で受け取り準備ができていないことを意味するため，マスタは同一のデータを時クロックサイクルでも同一のデータを送信する．また，一連のデータはパケットとして送受することになるが，最終データに対してはマスタから TLAST 信号も送信される．この TLAST 信号によりデータ転送のための DMA（Direct Memory Access）の終了が識別される．

4. 提案手法

本章では Pylog でストリーミング処理に対応させる手法について説明する．アプリケーションとして行列積（matmul）に注力して実装を行った．

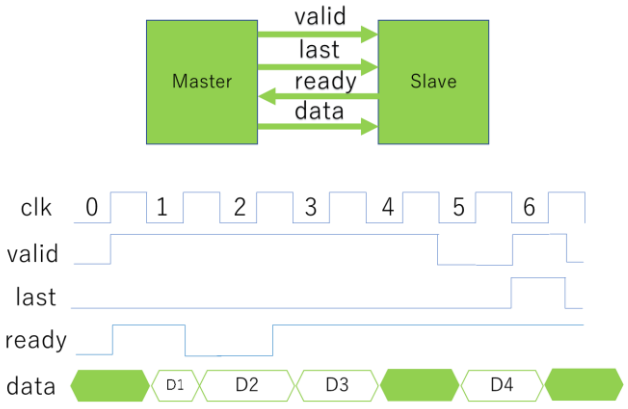

図 4 AXI4-Stream インタフェースの概要

Fig.4 Overview of AXI4-Stream Interface.

4.1 提案手法の概要

図 5 に提案手法の概要を示す．下記の流れで実装を行う．

（1） ストリーミング処理に対応した HLS C コード作成

●変数情報をソースコードから取得

●テンプレートを作成し変数情報を埋め込む

●ストリームファイルインタフェース定義

●行列データ型の定義

（2）ストリームインタフェースを呼び出す Vivado 向けテンプレートファイルを作成

●ストリームインタフェースの配置配線

●ビット幅等の回路設定

●テンプレートファイルの呼び出し，自動実行

これらのフローを基に実装を行っていく．

4.2 HLS C コード作成

図 6 の中間表現の一部より，args 変数で行列サイズを保存している．args[0]で行列 a に該当する情報を参照することができ，args[0].shape で行列の形，そして args[0].size で行列全体のサイズを取得することができる．これらの情報を他の関数で扱いやすいように変換しその情報を保持するようにした．

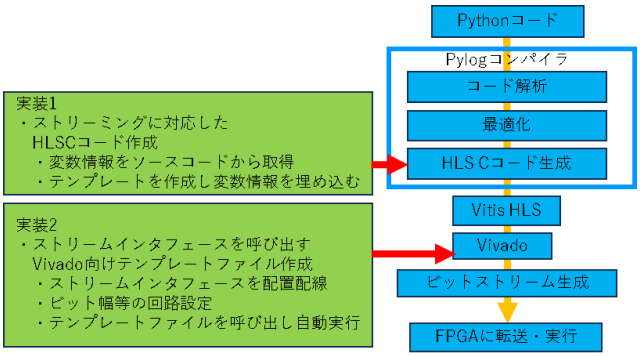

図 5 提案手法の概要

Fig.5 Overview of the proposed method.

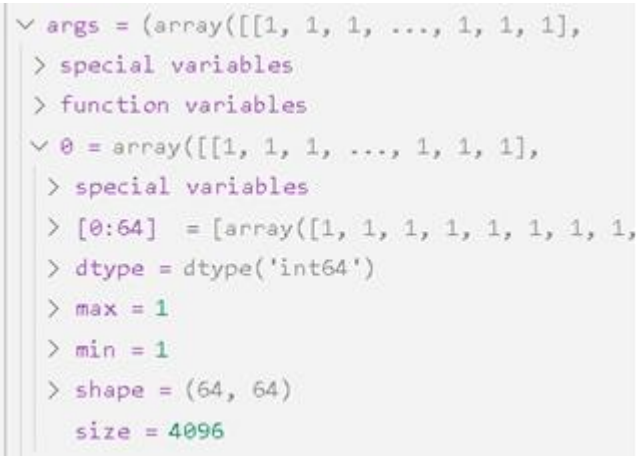

図 6 行列乗算の中間表現の例

Fig.6 Intermediate Representation of a matrix multiply.

ストリームインタフェースを呼び出し実装した際のコードの一部を図 7 に示す．hls::stream<axis>という形で定義する．また pragma を挿入しインタフェースの定義も行っている．

```
void matmul(
    hls::stream<axis_32_t> &a_stream,
    hls::stream<axis_32_t> &b_stream,
    hls::stream<axis_32_t> &c_stream
) {
#pragma HLS INTERFACE axis port=a_stream
#pragma HLS INTERFACE axis port=b_stream
#pragma HLS INTERFACE axis port=c_stream
#pragma HLS INTERFACE s_axilite port=return bundle=ctrl
```

図 7　プラグマによるストリームインタフェースの定義

Fig. 7　Pragma Directives for streaming interface.

4.3 ストリームインタフェースの実装

AMD/Xilinx 社の FPGA 設計ツールである Vivado を用いた FPGA 実装を行う場合，一般にユーザは合成を行う度に図 8 のようなブロックデザインを手動で作成する必要がある．

PyLog では Python で動作するテンプレートエンジンである“jinja2”を利用し，Vivado 用のスクリプトを生成し，それを用いて自動的に FPGA 実装が行われている．

jinja2 は構文{{ }}を用いて変数を記述することができ，これを利用することでプロジェクト毎に異なる値を反映させた実装スクリプトファイルを生成することができる．jinja2 のテンプレートから作成した Vivado 用スクリプトファイルの一部を図 9 に示す．プロジェクトの作成，IP 設定を自動化し，テンプレートファイルにパスや IP 名などの変数を埋め込み，インタフェースを配置配線する記述を行っている．ストリームインタフェースを利用する場合，AXI Direct Memory Access（AXI DMA）[6]と AXIInterconnect を図 9 下部のように create_bd_cell コマンドにより呼出している．その後，クロック信号やピンの接続を行う．配置配線を終えた後，validate_bd_design コマンドにより整合性の確認を取りビットストリームが生成される．

```
create_project {{ project_name }} {{ base_path }}/{{ project_name }} \
-part xc7z020clg400-1 -force

set_property board_part   tul.com.tw:pynq-z2:part0:1.0 [current_project]
set_property ip_repo_paths {{ ip_repo_path }} [current_project]
update_ip_catalog

create_bd_design "design_1"
update_compile_order -fileset sources_1

create_bd_cell -type ip -vlnv xilinx.com:ip:processing_system7:5.5 \
processing_system7_0

apply_bd_automation -rule xilinx.com:bd_rule:processing_system7 \
    -config {make_external "FIXED_IO, DDR" apply_board_preset {1}} \
    [get_bd_cells processing_system7_0]
set_property -dict [list CONFIG.PCW_USE_S_AXI_GP0 {1}] \
    [get_bd_cells processing_system7_0]

create_bd_cell -type ip -vlnv xilinx.com:hls:{{ ip_name }}:1.0 {{ ip_name }}_0

create_bd_cell -type ip -vlnv xilinx.com:ip:axi_dma:7.1 axi_dma_0
set_property -dict [list CONFIG.c_include_sg {0} ] [get_bd_cells /axi_dma_0]

create_bd_cell -type ip -vlnv xilinx.com:ip:axi_dma:7.1 axi_dma_1
set_property -dict [list CONFIG.c_include_s2mm {0} CONFIG.c_include_sg {0} ]\
[get_bd_cells /axi_dma_1]

create_bd_cell -type ip -vlnv xilinx.com:ip:axi_interconnect:2.1 axi_interconnect_0

validate_bd_design

save_bd_design

launch_runs impl_1 -to_step write_bitstream -jobs 4
```

図 9 Vivado 用合成スクリプト（*.tcl）の一部

Fig.9 Part of Vivadotcl file.

4.4 FPGA ボードでの動作

生成されたビットストリームを動作させるためには，ビットストリームファイルとハードウェア構成情報ファイルをボードに転送し，プロセッサ側で動作するプログラムの作成が必要である．合成により生成されたファイルの転送及び，実行ファイルの作成と起動が自動で動作できるように設計することでハードウェア合成から実行までの全てのフローを自動で実行することができる．ただし，本研究では転送及びプロセッサ側のファイル作成・実行は手作業で行った．

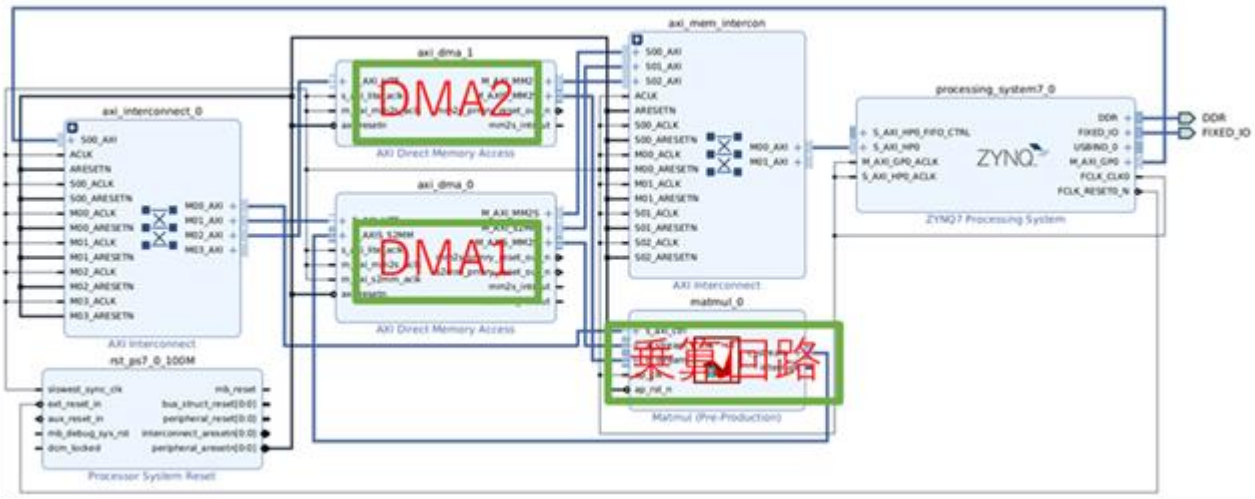

図 8 ブロックデザイン

Fig.8 Block Design.

5. 評価

今回提案した手法の有効性を確認するために，実行時間の評価と作成したツールの評価を行った．

5.1 評価条件

今回提案したストリームインタフェースを利用した実装と PyLog をそのまま利用した場合で比較を行う．

行列乗算の実行時間はストリーミング処理の適用による高速化の確認を行う．また，行列サイズを N×N サイズの正方行列で作成し，8×8 から 256×256 までの比較を行いサイズによるストリーミング処理の実行時間への影響を調査する．

行列乗算の比較対象は PyLog（AXIS），PyLog（M-AXI），NumPy の 3 種で実行時間の比較を行う．PyLog（AXIS）は提案手法であるストリーミング処理への対応を行ったもの，PyLog（M-AXI）は PyLog を従来のフローで利用した M-AXI インタフェースを利用したもの，NumPy は本研究で対象としている PYNQ-Z2 ボード上で NumPy を利用した CPU 処理の結果である．

ツールとしての評価では実装した機構を利用することでどのようなメリット・デメリットがあるか考察する．

評価環境について表 1 に示す．

表 1 評価環境
Table1 Evaluation environment.

ターゲットボード	Xilinx 社 PYNQ-Z2
ターゲットデバイス	Xilinx Zynq-7000 (XC7Z020)
計算機	Ubuntu 22.04.3 GNU/Linux 6.2.0-39-generic x86 64
高位合成ツール	Vitis HLS v2022.2
ハードウェア合成ツール	Vivado v2022.2

5.2 実行時間の評価

提案手法である AXI4-Stream の実行時間を AXIS，PyLog を従来のフローを利用した M-AXI での実行時間を M-AXI とする．そして PYNQ-Z2 ボード上で動作させた NumPy での実行時間を NumPy とする．表 2 に行列乗算を 10 回実行した際の平均実行時間を示す．

結果から，行列サイズが大きくなるにつれて，提案手法である AXIS の実行時間が他の実行方法に比べて有利であることが確認できる．すなわち，AXI4-Stream のデータ転送方式は行列サイズが大きい場合に効果的であることが確認できる．

一方で，行列サイズが 8×8 サイズと小さい場合は逆に AXIS の実行時間が M-AXI を利用した際の実行時間に劣る結果が得られた．これは AXI-DMA の起動等の処理がボトルネックとなっていることが考えられる．

表 2 行列乗算の実行時間
Table2 Execution times in the case of matrix multiplication.

行列サイズ	AXIS[ms]	M-AXI[ms]	NumPy[ms]
8×8	16.062	15.185	0.081
16×16	16.258	16.841	0.127
32×32	16.490	18.752	0.397
64×64	16.604	25.850	2.767
128×128	37.398	127.900	27.123
256×256	193.324	925.573	227.643

5.3 ツールとしての評価

実行時間の結果から，行列サイズが 8×8 と小さい場合には AXI-DMA の起動等の処理がボトルネックとなってしまっている．そのため，PyLog を従来フローで利用する方が実行時間の面においては適切であると考えられる．そこで，本実装では行列サイズが 8×8 より大きい場合に提案手法の適用を行うように生成フローに条件分岐を作成した．これによりユーザは特別な意識をする必要なく行列サイズが一定以上の時にストリームインタフェースを利用できるようになっている．一方で，自動で生成フローを切り替えるようにしているため，ユーザが使用するインタフェースを指定することができない設計となってしまっている．これに対しては，PyLog を実行する際のオプション等によりユーザがインタフェースを選択できるようにすることで対処できると考えている

なお，今回の実装では行列乗算に特化した実装を行ったため，他の計算をストリーミングで行う場合は別途設計が必要である．今後の課題として，行列乗算以外の演算機能においてもストリーミングインタフェースを適用できるような機能拡張を行っていきたいと考えている．

6. まとめ

本研究では，PyLog をストリーミング処理機構に対応させ，行列演算（matmul）の高速化を目指した．実装に当たって，行列サイズの抽出，ストリーミング処理に対応した HLS C コードの生成，及びハードウェア生成時のテンプレートファイルの作成を行った．その結果，従来の PyLog を用いた実装と比較して，行列サイズが大きくなった場合に実行時間が短縮され，ストリーミング処理機構の導入が性能向上に寄与することが確認できた．

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] 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
[5] PYNQ. 2021. [Online]. Available: http://www.pynq.io/
[6] AXI DMA [Online]. Aveilable: https://docs.amd.com/v/u/ja-JP/pg021 axi dma
[7] Xilinx, “Vivado high-level synthesis.” 2021.[Online]. Available: https://www.xilinx.com/products/design-tools/vivado/
[8] Falcon computing, “Merlin compiler.” 2021. [Online]. Available:https://www.falconcomputing. com/ merlin-fpga-compiler/
[9] J. Zhao, L. Feng, S. Sinha, W. Zhang, Y. Liang, and B. He,“Performance modelingand 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.
[10] axi stream T valid [Online]. Available: https://adaptivesupport.amd.com/s/question/0D52E00006hpNEvSAM/axi-stream-t-valid
[11] J. Dean, D. Patterson, and C. Young, “A new golden age in computer architecture: Empowering the machine-learning revolution,” IEEE Micro, vol. 38, no. 2, pp. 21–29, Mar./Apr. 2018.
[12] J. Ragan-Kelley, C. Barnes, A. Adams, S. Paris, F. Durand, and S. Amarasinghe, “Halide: A language and compiler for optimizing parallelism, locality, and recomputation in image processing pipelines,” SIGPLAN Not., vol. 48, no. 6, pp. 519–530, Jun. 2013.
[13] Y.-H. Lai et al., “HeteroCL: A multi-paradigm programming infrastructure for software-defined reconfigurable computing,” in Proc. ACM/SIGDA Int. Symp. Field-Programmable Gate Arrays, 2019, pp. 242–251.
[14] R. Nigam et al., “Predictable accelerator design with time-sensitive affine types,” in Proc. 41st ACM SIGPLAN Conf. Program. Lang. Des. Implementation, 2020, pp. 393–407.