深層学習モデルコンパイラにおける AI チップ用コード最適化

根岸 康　今井 晴基　Tung D. Le　河内谷 清久仁

深層学習モデルのビジネスにおける利用の普及に伴い，汎用 CPU に加えて深層学習専用演算器 (以下 AI チップ) を持つシステムが徐々に普及している．AI チップは GPU と同様深層学習の演算を高速に実行可能だが GPU とは異なる特性を持っている．GPU は通常全ての深層学習操作を効率的に実行可能であるため全ての操作が GPU 上で実行される．一方 AI チップは比較的小さな回路で実装されるため，全深層学習操作をサポートせず，AI チップ上で効率的に実行可能な一部の深層学習操作のみをサポートすることが多い．これはほとんどの深層学習モデルでは Convolution/LSTM 等小数の深層学習操作が実行時間の大半を占める状況に適応している．また，AI チップは通常 bfloat16/DLFloat16 等深層学習専用の浮動小数点形式や独自のデータ構造のみをサポートし，CPU は通常浮動小数点形式をサポートするので実行切り替え時に入出力データの表現形式を変換する必要がある．このため深層学習モデルに際しては必要に応じて CPU・AI チップ間の実行を切り替える必要があり，深層学習モデルの実行環境は CPU と AI チップの性能に加えて切り替えコストにも留意して，各操作を CPU 上で実行するか AI チップ上で実行するかを決定する必要がある．本稿では深層学習モデルの実行環境の一例として onnx 形式の深層学習モデルを入力とし推論用の実行可能コードを生成する深層学習モデルコンパイラ，具体的には我々が開発中の onnx-mlir コンパイラ及び z16 システム Telum プロセッサの AIU チップを想定して，各深層学習操作の実行装置の決定手順について考察する．実行装置は深層学習操作毎に独立に決定可能なので，その際の理論的な探索空間は 2 の「深層学習操作の数」乗となるが，モデル内には多数の深層学習操作が含まれているため全ての組み合わせをブルートフォースで探索することは現実的でない．深層学習モデルコンパイラがこの探索空間を効率的に検索して，最適解を得るために，本稿では (1) 幾つかのヒューリスティクスを用いて問題空間を分割・削減する手法，(2) 分割後の問題空間を有向グラフに変換してグラフ理論の最大フロー最小カットの問題に帰着する手法，(3) その実装方法について提案・考察する．

1　はじめに

深層学習モデルはビジネスの様々な業務で使用され始めている．深層学習モデルの実行には多くの場合 GPGPU [4] と呼ばれる画像用プロセッサを拡張した演算器が用いられるが，近年深層学習固有の演算に特化した専用回路を持つ深層学習演算専用演算器 (以下 AI チップと呼ぶ) [4] [1] [3] [2] が普及し始めている．AI チップの目的はさまざまで，学習フェーズで GPGPU を超える演算性能や消費電力性能を求めるものや，業務アプリケーションの実行環境で推論フェーズの性能を向上させる目的のもの等がある．AI チップを利用するには，PyTorch [10]，TensorFlow [11] 等の既存の深層学習用フレームワークを拡張する方法，学習済み深層学習モデルから深層学習コンパイラ [5] で実行コードを生成する方法等がある．本稿では主に深層学習コンパイラを用いる方法を前提として説明するが，提案手法は利用方法によらず適用可能である．以下，第 2 章で背景となる AI チップと深層学習コンパイラについて，第 3 章で提案手法について説明する．第 4 章で提案手法の効果について考察し，第 5 章でまとめと今後の課題について述べる．

2　背景

本章では，AI チップ及び AI チップ向けコードを生成する深層学習コンパイラについて説明する．

An Optimization for AI Chip by Deep Learning Model Compiler

Yasushi Negishi, Haruki Imai, Tung D. Le, Kiyokuni Kawachiya, 日本アイ・ビー・エム株式会社 東京基礎研究所, IBM Research - Tokyo.

2.1 AI チップ

本節では AI チップと呼ばれる深層学習演算専用回路を持つシステムの例を説明する．

2.1.1 AIU

IBM 社のホストマシン z16 の Telum プロセッサには AIU [1] とよばれる深層学習推論用 AI チップが付加されている（図 1 参照）．利用形態として主にホストアプリケーションによる深層学習推論実行を加速化することを想定しており，この回路によりホスト上のアプリケーションがクレジットカード決済等のトランザクション実行時に深層学習を使って不正検知を行う等の処理をホストのセキュリティーレベルを守ったまま容易に行える．z16 Telum プロセッサは 8 つの 5+GHz の CPU コアを持ち，これらが 2 つの AIU コアを共有する．各 AIU コアは以下の 2 つの演算器を持ち，L2 キャッシュ上のデータにアクセスする

1. 行列積・Convolution 操作用マトリックスアレイ 8 way FP-16 SIMD で 6TFlops 以上の計算能力を持つ 128 プロセッサタイルを持つ
2. Activation 関数・LSTM/GRU 操作用 Activation アレイ 8 way FP16/FP32 SIMD で 32 プロセッサタイルを持つ

これらの演算器は DLFloat16 と呼ばれる深層学習向け浮動小数点形式と演算器専用の行列表現をサポートする．CPU は標準的な IEEE754 32 ビット浮動小数点と行列表現を使用するので AIU の使用時は入出力データ及び中間データの浮動小数点及び行列の形式を必要に応じて変換する必要がある．学習済モデルが持つ重み・バイアス等のパラメータは必要に応じてコンパイル時に形式変換可能なので実行時の変換は通常の場合不要である．AIU の機能は zDNN ライブラリと呼ばれる API を経由して利用する．行列積・Convolution 操作，Activation 関数・LSTM/GRU 操作以外に 2 つの行列の和・差等を要素毎に計算する element-wise 操作，Max Pool 操作等がサポートされる．

2.1.2 TPU

Tensor Processing Unit(TPU) [3] は Google により開発された深層学習演算専用回路で機械学習ワークロードの高速化に使用される．TPU は TensorFlow, Pytorch，JAX 等の機械学習フレームワークを通して利用可能であり，bfloat16 と呼ばれる深層学習用の 16 ビット浮動小数点をサポートしている．チップ当たりの計算能力は 275 TFlops(bfloat15 もしくは int8) である．メモリとして最大 32GiB の HBM2 を使用する．

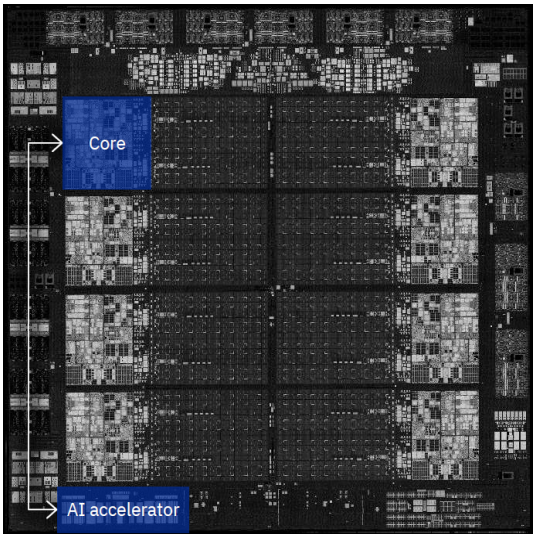

図 1 IBM Telum プロセッサと AIU

2.1.3 MN-Core2

MN-Core [2] は Preferred Networks により開発された深層学習専用チップで，深層学習に用いられる行列演算に特化した計算をサポートする．MN-Core を利用したシステムは 2020-2021 年に Green500 の一位を獲得しており，2024 年度には次世代のチップ MN-Core2 を利用したスーパーコンピュータ MN-4 を構築する予定である．

2.2 深層学習モデル実行環境

本節では深層学習モデルを入力として対象マシン上での実行をサポートする深層学習モデル実行環境について，onnx-mlir コンパイラ [5] と onnx-runtime [8] 実行環境を例にとって説明する．どちらも onnx 形式の学習済みモデルを入力として，対象マシン上の実行コードを生成するツールである．

2.2.1 Onnx 形式学習済み深層学習モデル

onnx は，学習済み深層学習モデルを交換するための標準ファイル形式及びそのエコシステムであり，関連ソフトウェアがオープンソースとして公開されている [7]．学習フェーズ用ソフトウェアと推論フェーズ用ソフトウェアは onnx 形式ファイルを経由して学習済みモデルを交換することが可能になる (図 2 参照)．

- Onnx 形式ではオペレータ (操作) と標準データ形式を含むモデルの計算グラフを定義し，現状主に推論フェーズの実行をサポートする．
- TensorFlow，Keras，PyTorch，MATLAB 等多数のプラットフォームが onnx 形式ファイルの出力をサポートする．
- onnx 形式の学習済みモデル及びその入出力データは model zoo と呼ばれるサイト [6] で公開されている．

2.2.2 Onnx-mlir

Onnx-mlir は onnx 形式の学習済み深層学習モデルを入力として，対象マシン上で動作する実行コードを生成するコンパイラで，OS として Windows，Linux，MacOS，プロセッサとして X86，Power Processor，Telum，AI チップとして AIU をサポートする (図 2 参照)．onnx-mlir コンパイラは LLVM コンパイラ基盤に基づいて実装されている．Onnx-mlir の開発には IBM，Microsoft，Arm，Facebook 等の企業が参加している．デザイン及び実装目標は以下の通り．

- ONNX モデルの MLIR/LLVM 形式 Dialect の参照実装となる．
- 高レベル (グラフレベル) から低レベル (命令レベル) の CPU と専用アクセラレータ向けコード最適化をサポートする．
- Python/C/C++/Java で記述された独立したドライバとランタイムで簡単に導入可能にする．
- x86/Power/Telum 等の多様なプロセッサ，Windows/Linux/macOS/zOS 等の様々な OS での継続的なテストを実施する．

2.2.3 Onnx-runtime

Onnx-runtime [8] は ONNX モデルを運用環境にデプロイするための高性能な実行環境で，Linux，Windows，Mac，Android，iOS 等 OS，TPU，GPU 等のアクセラレータをサポートしている．

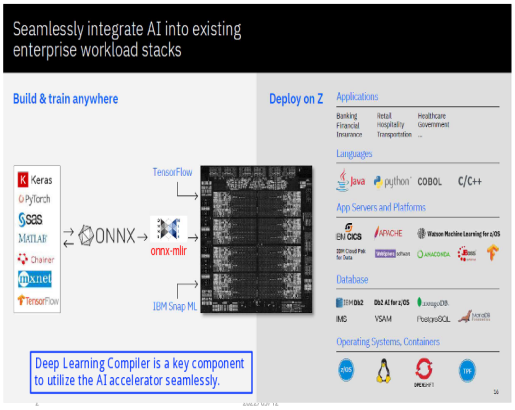

図 2　Onnx-mlir の入出力

3　提案手法

提案手法は AI チップ用コード生成に向けて onnx モデルの各操作を CPU もしくは AI チップのどちらで実行すべきかを決定するためのものである．本章で提案手法について説明する．

3.1　提案手法の技術課題

前章で述べた通り AI チップは AI チップが効率的に実行可能な一部の深層学習操作のみをサポートする．このため深層学習モデルに際しては必要に応じて CPU・AI チップ間の実行を切り替える必要がある．実行切り替えには，浮動小数点表現形式の変換等一定のコストがかかるため，深層学習モデルの実行環境は CPU と AI チップでの実行時間に加えて切り替えコストにも留意して，モデル中の全操作について CPU と AI チップのどちらで実行するかの組み合わせを決定する必要がある．

3.2　提案手法の前提条件

決定に際しては，対象モデルの全操作の CPU 上及び AI チップ上での実行時間，データ変換に必要な実行時間がコンパイル時に判明していることを前提とす

る．各操作やデータ変換に必要な時間は同一のモデルを CPU 及び AI チップで事前に実行することで取得してもよいし，各操作のパラメータを元に実行時間やデータ変換時間を予測する予測モデルを使用してもよい．深層学習モデルにはバッチ数等一部のパラメータの値が固定されないものもあるが，事前に適当な予測値で固定し，コンパイル時に全てのパラメータがコンパイル時に決定されているものとして説明する．バッチ数等ほぼ全ての操作の処理時間に単純に比例するパラメータが変動する場合装置決定に影響ない．そうでない場合は複数の予測値でコンパイル・コード生成し，実行時には近い値で生成されたコードを選択する等の工夫が必要となることも考えられるが，本稿では触れない．また AI チップは CPU と比較して精度の低い浮動小数点表現を用いていることが多く，実行環境等がその影響について考慮する必要がある場合があるが，本稿では触れない深層学習モデル全体の実行時間は，その全操作について CPU と AI チップのどちらで実行するかの組み合わせ（以下，実行装置決定表と呼ぶ）が与えられれば上記の各操作の実行時間と必要なデータ変換時間の和として計算可能となる．

3.3 提案手法の基本方針

学習モデルの実行環境は，実行装置の全ての組み合わせについてモデル全体の実行時間を計算し，それが最小となる CPU と AI チップの組み合わせを求めればよいが，この時に課題となるのが探索空間の大きさである．全操作について CPU もしくは AI チップで行うかを単純に探索すると「2 の (操作数) 乗」となり，現実的な探索は難しくなる．本論文では，ヒューリスティクスやグラフ理論の手法により，探索空間を分割・削減し，現実的なコストで実用的な近似解を求める方法を提案する．提案手法の妥当性については手順の説明後に説明する．

3.4 提案手法の手順

本節では提案手法の手順について説明する．手順は大きく (1) 入力 onnx モデルの計算グラフからコストグラフを作成し，(2) コストグラフから装置決定グラフを作成し，(3) 装置決定グラフから各操作に使用する装置を決定するという流れとなり，以下の 6 つのステップから構成される．

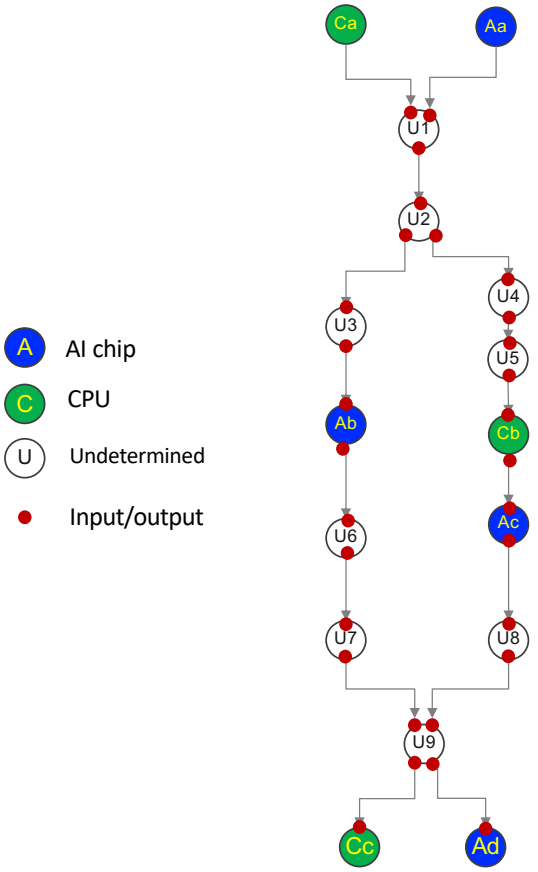

図 3　計算グラフ

(STEP1) 操作の分類

Onnx モデルは操作とその入出力から構成される計算グラフ (図 3 参照) により表現される．モデルの各 onnx 操作を次の 3 種類に分類し，AI チップ操作は AI チップ上で，CPU 操作は CPU 上実行することを決定する．以後，未確定操作のみ実行装置の探索を進める．

AI チップ操作

AI チップにより効率的に実行可能な操作．AIU の場合 Convolution 操作，LSTM/GRU 操作，行列積操作が相当する．我々の AIU の経験によれば，ほとんどの深層学習モデルでこれらの操作が実行時間の大半を占めており，AIU により数倍以上の性能向上が得られるため，ほとんど常に

AI チップ上での実行が最適となる．

CPU 操作

AI チップでサポートされず CPU 上でのみ実行可能な操作．AIU の場合，Split 操作，Concat 操作，Gather 操作，Transpose 操作，Unique 操作等が相当する．また，AIU の場合対象 Tensor の大きさを自動的に合わせる broadcast 必要な操作も相当する．これらの操作は CPU でのみ実行可能となる．

未確定操作

CPU 上でも AI チップ上でも実行可能な操作．AIU の場合，Element-wise 操作，Average Pool 操作，Max Pool 等が相当する．多くの場合処理時間が CPU・AI チップ間データ変換処理時間より短いため，データ変換コストに留意して実行装置を決定する必要がある．

(STEP2) 計算グラフの分割

STEP1 での計算グラフの各ノードの分類に基づいて，CPU ノードもしくは AI チップノードで，計算グラフを分割する．分割した点となったノードは分割された計算グラフの両方に含まれる．以下の処理は分割された計算サブグラフ毎に行う．

(STEP3) コストグラフの作成

提案手法では，分割された計算サブグラフ毎に下記の手順でノード及びエッジに重みをつけ，ノードとエッジの両方に重みを持つ有向グラフに変換する．具体的には計算サブグラフの各ノードに対応する操作の CPU と AI チップでの推定処理時間の 2 つを重みとして与え，各エッジに対応するそのエッジの前後で計算装置が切り替えられた場合に必要なデータ変換処理時間を重みとして与える．以下，この手順で生成されたエッジを変換コストエッジと呼び，生成されたグラフをコストグラフと呼ぶ．図 4 は生成されたコストグラフの一例を示す．図中 Tc1,Tc2... は CPU を使用した時の操作実行時間，Ta1, Ta2... は AI チップを使用した時の操作実行時間を，D12，D23... は操作間にデータ変換が必要であった場合のデータ変換時間を示す．

(STEP4) 未確定操作の処理装置の確定

提案手法では作成されたコストサブグラフ毎にそれぞれ以下の処理を行う．

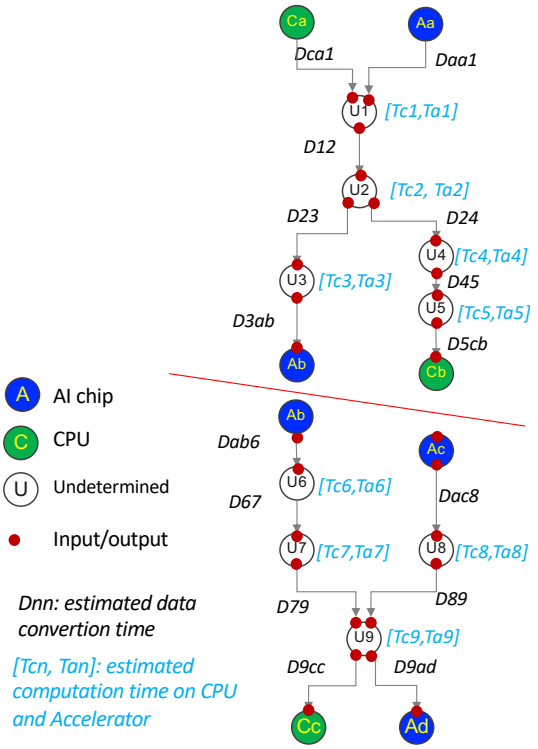

図 4　コストグラフ

1. コストサブグラフに AI チップ操作が含まれない場合，グラフの全ての操作を CPU 操作とする．
2. コストサブグラフに CPU 操作が含まれない場合，グラフの全ての操作を AI チップ操作とする．
3. コストサブグラフに AI チップ操作と CPU 操作の両方が含まれる場合は STEP5 に進む．

(STEP5) コストグラフから装置決定グラフへの変換

コストグラフから装置決定グラフへの変換は以下の手順で行う．

装置決定グラフのノードの生成

コストサブグラフ中の全ての CPU 操作を一つのノードに集約して生成し，集約したノードを集約 CPU ノードと呼ぶ．また同様に全ての AI チップ操作を一つのノードに集約して生成し，生成したノードを集約 AI チップノードと呼ぶ．また，未確定操作に対応するノードはそのまま残す．

装置決定グラフのエッジの生成

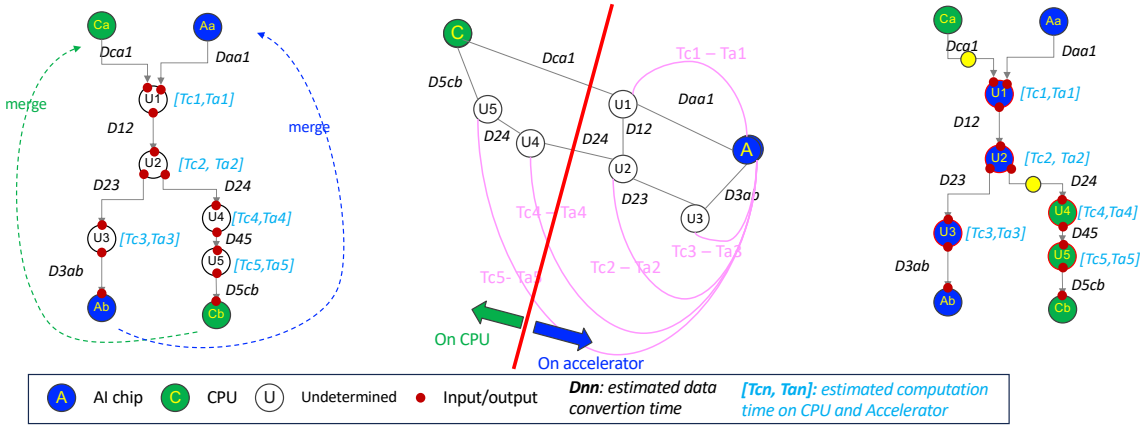

図 5　コストグラフ (左)，装置決定グラフ (中)，最終結果の計算グラフ (右)

コストサブグラフのエッジは重みも含めてそのまま装置決定グラフのエッジとして生成する（以下このエッジを変換コストエッジと呼ぶ）．更に各操作でＣＰＵによる処理時間から AI チップによる処理時間を引いた時間を重みとして持つ各操作ノードから集約 AI チップノードへのエッジを追加する（以下このエッジを計算コストエッジと呼ぶ）．また，生成されたブラフを装置決定グラフと呼ぶ．

(STEP6) 最小カット・最大フロー手法による装置の決定

1. 作成した装置決定グラフの集約 CPU ノードと集約 AI チップノード間のフローに対して，グラフ理論の最小カット・最大フロー手法を適用し，装置決定グラフを 2 つに切断する．
2. 切断された 2 つのグラフ中の内 CPU ノードを含むグラフの全てのノードを CPU 操作と決定し，集約 AI チップノードを含むグラフの全てのノードを AI チップ操作と決定する．

ここまでのステップで，全ての操作を CPU 操作もしくは AI チップ操作と決定できる．

3.5　提案手法の妥当性

本節では提案手法の解及び計算時間の妥当性について説明する

3.5.1　求めた最適解の妥当性

STEP6 で生成された装置決定グラフの任意の断面は，装置の決定組み合わせの一つに対応し，そのコストの和はその決定に対応する実行時間に相当する．この内変換コストエッジの断面の和がモデル全体の変換コストの和に相当し，計算コストエッジの断面の和がモデル全体の全ての AI チップで計算した場合と比べての計算コストの増加分に相当する．このため装置決定グラフをグラフ理論の最小カット・最大フロー手法で分割することは，この断面すなわちモデル全体の実行時間を最小化する事に相当する．

3.5.2　計算時間の妥当性

提案手法で使用するグラフ理論の最小カット・最大フローの決定手順には様々な手法が提案されており，[9] によれば n 頂点 m 辺のグラフに対する計算量は O(nm) であり，元々の探索空間と比較するとずっと小さく，数万〜数百万のオーダーの問題に対しても実用的な時間で適用可能であることが知られている．

3.6　提案手法の実装方針

提案手法の実装に際しては，事前に実行環境での各操作の推定実行時間等が必要なことから，onnx-mlir コンパイラとは独立したツールとして実装する予定である．利用者は，これまでの onnx-mlir コンパイラへのコマンドオプションに加えて，実行に必要な入出力データについての引数をこのツールに渡すことで，onnx-mlir と同様に提案手法のツールを使用する

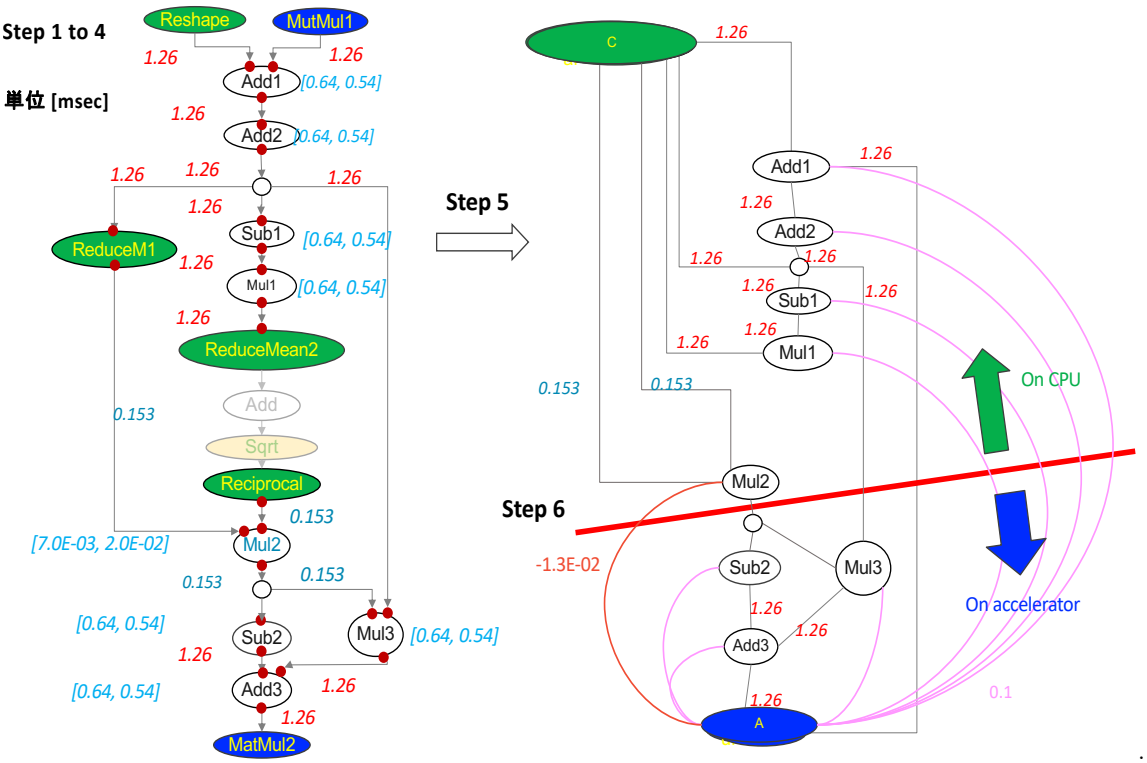

図 6　想定する深層学習モデルのコストグラフ (左) と装置決定グラフ (右)

ことが可能となる．ただし，コンパイルしたコードを対象マシン上で実行し各操作の実行時間を測定するために，対象マシン上で提案手法のツールを使用する必要がある．ツールの実装方針は以下の通り．

1. 与えられた深層学習モデルを (1) AI チップ不使用 (2) AI チップ最大使用の 2 つの方法でコンパイルする．
2. コンパイルした 2 種類のコードをターゲットノード上で実行し，各オペレータの CPU 及び AI チップ上での各操作の実行時間を測定する．この 2 回の測定で全操作の CPU 及び AI チップ上での実行時間が測定できる．データ変換操作の実行時間は想定されるデータ量から推定する．
3. 深層学習モデル及び測定した実行時間を入力として提案手法を実行し，各ノードを CPU もしくは AI チップのどちらで実行すべきかを決定する．
4. 現在の onnx-mlir コンパイラは標準で AI チップを利用可能な操作全てを AI チップ上で実行するが，指定した操作を CPU 上で実行する“–execNodesOnCpu”オプションを持つ．提案手法で決定された CPU 操作をこのオプションで指定してコンパイルすることで，提案手法で決定した装置での実行を指定することが可能である．

4　提案手法に期待される効果と考察

現在の onnx-mlir コンパイラは AI チップ上で実行できるすべての操作 AI チップで実行する方針を採用している．一方提案手法ではデータ変換コストを考慮して各操作の実行装置を決定する．本章では，一つの学習モデルを想定して提案手法を onnx-mlir に使用した場合に期待される実行時間を推定し，現状の onnx-mlir コンパイラを使用した場合実行時間と比較する．また，また比較結果について考察する．

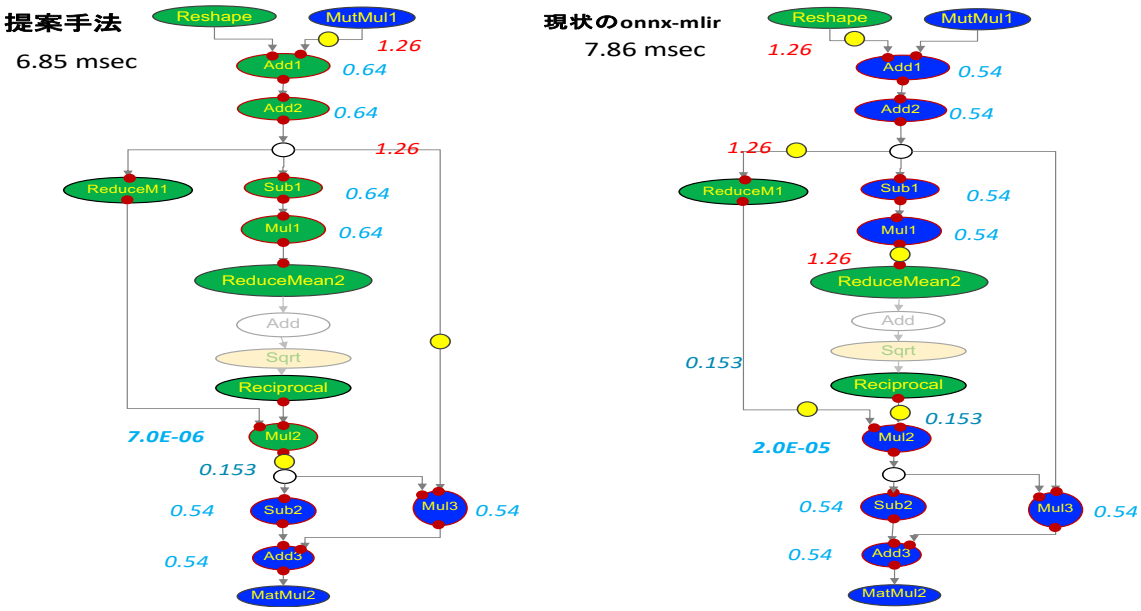

図 7　提案手法の適用結果

4.1　想定する深層学習モデル

以下，図 6 で示される計算グラフで表現される深層学習モデルを想定して，現状の onnx-mlir コンパイラを使用した場合及び提案手法を適用した場合の実行時間を推定する．この例では CPU 操作として Reshape, Reduce，ReduceMean，Pciprocal 操作，AI チップ操作として MatMal 操作，未確定操作として Add, Sub，Mul 操作を含むモデルを想定している．各操作の実行時間は z16 Telum CPU 及び AIU での実行時間の実測に基づいたもの，データ変換時間は実測に基づいてデータ量から推定して得られたものである．

4.2　提案手法の適用

上記モデルに対して提案手法を適用する．提案手法はまだ実装されていないので，適用結果は筆者らが手動で提案手法に基づいた変換を行った結果である．以下，提案手法により決定された最終的に図 7 の結果を得る．現状の onnx-mlir コンパイラを使用した場合の推定実行時間は 7.86 ミリ秒，提案手法を使用した場合の推定実行時間は 6.85 ミリ秒となった．

4.3　考察

現状の onnx-mlir コンパイラと提案手法の適用結果を比較すると，現状の onnx-mlir コンパイラで AI チップを使用していた操作の一部が提案手法によれば CPU 上で実行されている．これにより CPU から AI チップ及び AI チップから CPU への変換コストが削減されて，モデル全体の実行時間も削減されていることが確認できた．

5　まとめと今後の課題

本稿では深層学習モデルの実行環境の一例として onnx 形式の深層学習モデルを入力とし推論用の実行可能コードを生成する深層学習モデルコンパイラ，具体的には実行環境として我々が開発中の onnx-mlir コンパイラ及び AI チップとして z16 システム Telum プロセッサの AIU を想定して，各深層学習操作の実行装置の決定手順について考察した．実行装置は深層学習操作毎に独立に決定可能なので，その際の理論的な探索空間は 2 の「深層学習操作の数」乗となるが，モデル内には多数の深層学習操作が含まれているため全ての組み合わせをブルートフォースで探索することは現実的でない．深層学習モデルコンパイラが

この探索空間を効率的に検索して，最適解を得るために，本稿では (1) 幾つかのヒューリスティクスを用いて問題空間を分割・削減する手法，(2) 分割後の問題空間を有向グラフに変換してグラフ理論の最大フロー最小カットの問題に帰着する手法，(3) その実装方法について提案・考察した．今後の課題は，提案手法を onnx-mlir 向けプリプロセッサとして実装して，より多くの実際的なモデルでの性能の向上を確認することである．

謝辞 本論文の作成に際して，onnx-mlir の開発についての議論を続けている共同開発者の皆様に感謝いたします．

参 考 文 献

[1] IBM: IBM z16, https://www.ibm.com/jp-ja/products/z16.
[2] Networks, P.: MN-Core Series, Deep Learning Accelerators, https://projects.preferred.jp/mn-core/.
[3] Norman P. Jouppi, e.: TPU v4: An Optically Reconfigurable Supercomputer for Machine Learning with Hardware Support for Embedding.
[4] Nvidia: NVIDIA TESLA V100 GPU ARCHITECTURE, https://images.nvidia.com/content/Volta-architecture/pdf/Volta-architecture-whitepa pdf.
[5] onnx-mlir project, T.: Onnx-mlir, https://github.com/onnx/onnx-mlir.
[6] onnx model zoo project, T.: ONNX Model Zoo, https://github.com/onnx/models.
[7] onnx project, T.: Onnx, https://github.com/onnx.
[8] onnx-runtime project, T.: Onnx-runtime, https://github.com/microsoft/onnxruntime.
[9] Orlin, J. B.: TMax flows in O(nm) time, or better, *STOC '13: Proceedings of the forty-fifth annual ACM symposium on Theory of Computing*, ACM, 2013.
[10] project, T. P.: PyTorch, https://github.com/pytorch/pytorch.
[11] project, T. T.: TensorFlow, https://github.com/tensorflow/tensorflow.

A MLIR/LLVM コンパイラインフラストラクチャ

onnx-mlir は MLIR/LLVM コンパイラインフラストラクチャ上に実装されている．Onnx 形式のファイルは，MLIR/LLVM で定義される Dialect と呼ばれる中間言語に変換され，最終的に実行コードに変換される．onnx-mlir では，AIU 向けコード生成のために ZHigh Dialect/ZLow Dialect の 2 つの Dialect を定義・使用している．図 8 は onnx-mlir が使用する Dialect 間の変換の概要を示す．黄色で示された部分が AIU のための部分である．また，図 9 に中間言語への変換例を示す．

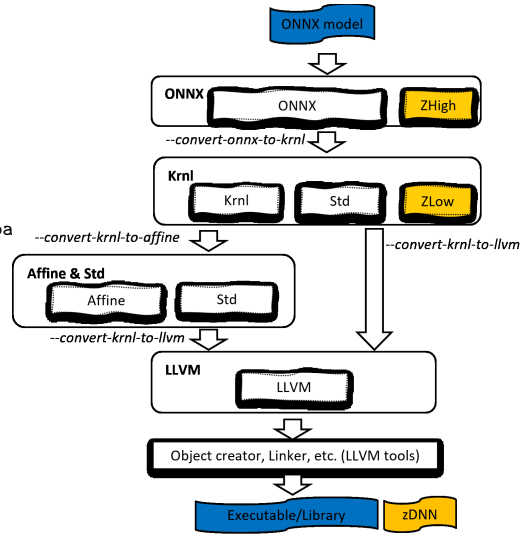

図 8　onnx-mlir の AIU 向け Dialect

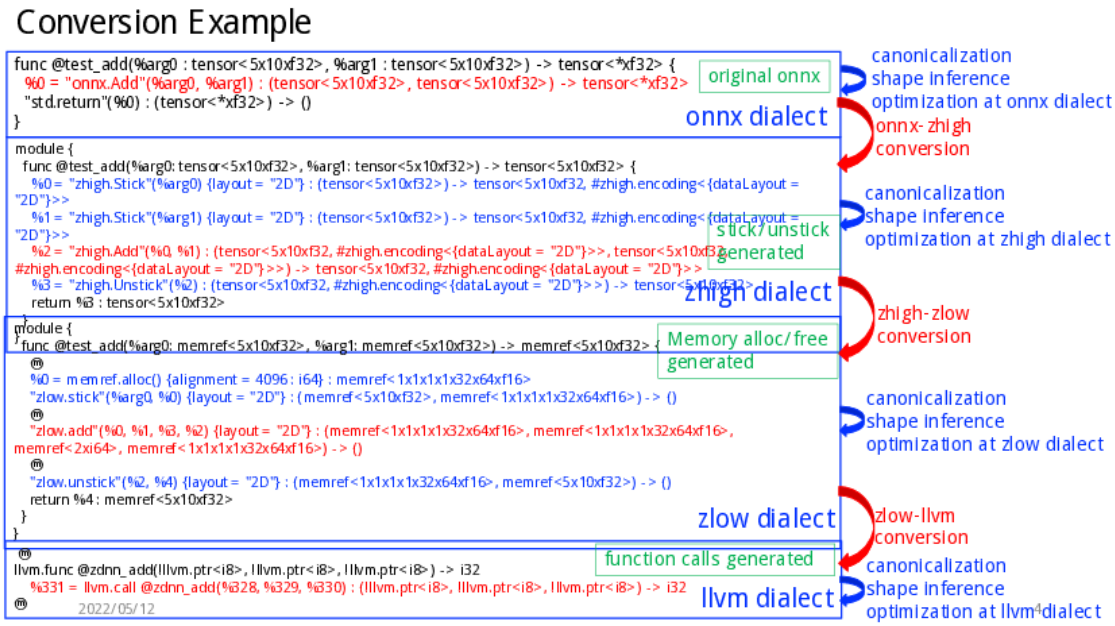
Conversion Example
original onnx
onnx dialect
canonicalization
shape inference
optimization at onnx dialect
onnx- zhigh
conversion
stick/unstick
generated
zhigh dialect
canonicalization
shape inference
optimization at zhigh dialect
zhigh-zlow
conversion
Memory alloc/free
generated
zlow dialect
canonicalization
shape inference
optimization at zlow dialect
zlow-llvm
conversion
function calls generated
llvm dialect
canonicalization
shape inference
optimization at llvm dialect
2022/05/12

図 9　変換例