Vol.2023-ARC-254 No.8
2023/8/3

深層学習コンパイラTVMの ベクトルマルチコア向けコード生成手法の検討

大西 文彬[1,a)]　大髙 凌聖[1]　藤田 一輝[1]　末次 智貴[1]　川角 冬馬[1]　北村 俊明[1]　笠原 博徳[1]
木村 啓二[1]

概要：自動運転車やスマートロボットなどのIoTデバイスのような組み込み機器においても，高度な判断や制御をするために，深層学習による推論処理が広く利用されつつある．しかしながら，画像入力に対する推論処理で多用される畳み込み演算は計算量が大きく，消費電力の増大とそれによる発熱量の増大を招いてしまい，その結果組み込み機器に大容量のバッテリーが必要となったり，冷却装置のための大きなスペースが必要となったりする懸念がある．これはスマートロボットなどにおいては行動自由度の低下や筐体の大きさにも影響を与える．これに対して筆者等は，コンパイラ協調のOSCARベクトルマルチコアによる，深層学習処理の高速かつ低消費電力実行の実現を目指している．本マルチコアでは，既存の多くの学習モデルを利用すべく，そのコンパイルツールチェーンに深層学習コンパイラTVMを取り入れる．本稿ではOSCAR自動並列化コンパイラとTVMによるベクトルマルチコア用コンパイルツールチェーンの有効性を示すべく，TVMによるベクトルマルチコア用コード生成手法を提案・実装する．さらに提案手法を実装したTVMをOSCARコンパイラおよびNECを組み合わせ，ResNetの学習済み深層学習モデルをNECのベクトルマルチコアであるSX-Aurora TSUBASA上で評価した．評価では上記TVMの生成コードをOSCARコンパイラで並列化し，さらにNECコンパイラによりベクトル化した．評価の結果，提案手法実装前後の同コア実行時の実行時間を比較すると，1コア実行時に14.0倍，8コア時に27.1倍の速度向上をそれぞれ得た．

Investigation of code generation techniques for vector multicore targeting using the deep learning compiler TVM

Fumiaki Onishi[1,a)]　Ryosei Otaka[1]　Kazuki Fujita[1]　Tomoki Suetsugu[1]　Tohma Kawasumi[1]
Toshiaki Kitamura[1]　Hironori Kasahara[1]　Keiji Kimura[1]

1. はじめに

自動運転車やスマートロボット等，組み込み機器においても深層学習による画像認識や音声認識をはじめとする高度な処理を要求するものが増えている．このような深層学習の処理には，多数の配列の畳み込み演算などにより，大きな計算コストがかかる．

このような処理を高速に行う方法として，多数の高速な演算器や高バンド幅のメモリの導入が挙げられるが，処理の余裕度を確保するために対象となるアプリケーションの要求性能を過剰に超過するハードウェア資源の導入は不要な消費電力の増大やそれに伴う発熱を招き，大容量バッテリーや大きな冷却装置の導入を必要とする．IoTデバイスを始めとする多くの組み込み機器では筐体の大きさや機材設置の自由度の観点から，これらの大消費電力や高発熱に伴うハードウェアフットプリントの増大は最小限とすべき項目である．すなわち，とりわけスマートロボット等の高機能組み込み機器における高性能・高電力効率のコンピュータシステムの導入が肝要である．

高性能・高電力効率のシステムを実現する方法として，

[1] 早稲田大学
Waseda University
[a)] nueshitone@kasahara.cs.waseda.ac.jp

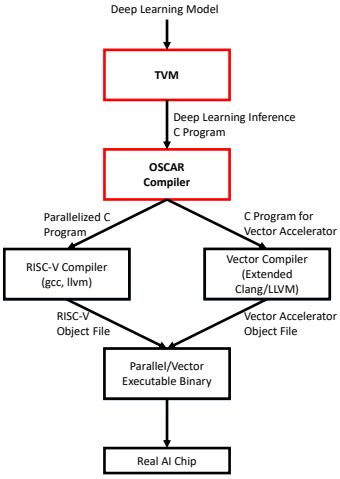

図 1 TVM と OSCAR を用いた深層学習モデルのコンパイルフロー

プログラムを低電圧かつ低動作周波数の複数コアを持つマシン上で，電圧や動作周波数を適切に設定しながら並列に実行することで，高動作周波数のマシン上で実行した場合以上の性能を達成する方式が提案されている [1]．このような低消費電力なコンピュータシステムを実現するためには，対象アプリケーションの挙動に応じて適切なコア数と動作周波数及び電圧等を細かく制御する自動並列化コンパイラとマルチコアアーキテクチャの協調が必要である．筆者等は，各コアが，グローバル空間にマップされ分散共有メモリとしても使用されるローカルメモリとベクトルアクセラレータを持つ OSCAR ベクトルマルチコアと，それに対応するためのベクトル化や並列化，メモリ最適化，電力最適化を行う OSCAR 自動並列化コンパイラを開発している [2]．ベクトルアーキテクチャでは，1 つの命令で複数のデータ要素を同時に処理できるため，OSCAR ベクトルマルチコアはデータ並列性の高い処理に対して特に優れた性能を発揮する．

一方，深層学習のアプリケーション開発には Tensorflow[3] や PyTorch[4] などのフレームワークを用いることが多い．これらのフレームワークによる深層学習モデルを共通フォーマットである ONNX[5] を介して深層学習コンパイラ TVM[6] で処理することにより，OSCAR 自動並列化コンパイラで並列化可能な C プログラムを生成できる [2]．すなわち，フレームワークを用いた豊富な学習モデルを OSCAR 自動並列化コンパイラと OSCAR ベクトルマルチコアで利用可能となる．このような深層学習モデルのコンパイルフローを図 1 に示す．

このコンパイルフローでは，Tensorflow や PyTorch などの様々なフレームワークを用いて学習された機械学習モデルを，深層学習コンパイラ TVM を用いて逐次 C ソースコードに変換し，OSCAR 自動並列化コンパイラに入力する．OSCAR 自動並列化コンパイラでは，ホスト CPU に向けた並列化 C ソースコードと，ベクトルアクセラレータに向けた C ソースコードを出力する．最後にこれらの C ソースコードを用いて実行バイナリを生成する．このように，TVM を用いることで機械学習モデルを OSCAR 自動並列化コンパイラに入力することができる．

しかしながら，現在の TVM はベクトル化に適した再内側ループが十分に長いコードを生成せず，その生成コードはベクトル化効率が低い．そこで本稿では，学習モデル中のテンソルのレイアウトを変形し，その生成コードがベクトル化に適した十分な長さの再内側ループとなるようなコード生成手法を提案する．さらに提案手法を実装した TVM を OSCAR 自動並列化コンパイラと組み合わせて NEC のベクトルマルチコアである SX-Aurora TSUBASA[7] 上で評価した結果について報告する．本評価では図 1 に沿って，TVM が生成した学習モデルのコードを OSCAR 自動並列化コンパイラの入力としマルチコア並列化を適用する．さらに，並列化したコードを NEC コンパイラに入力しベクトル最適化を適用し実行バイナリを生成した．

以下，2 節で OSCAR 自動並列化コンパイラについて，3 節で TVM の概要について説明し，4 節でコード生成と高速化のための実装について述べる．5 節で性能評価の結果について報告し，6 節でまとめる．

2. OSCAR 自動並列化コンパイラ

OSCAR 自動並列化コンパイラは，C 言語あるいは Fortran で書かれた逐次プログラムをソースコードとし，OpenMP と互換性のある OSCAR API[8] による並列化コードを生成する Source to Source の自動並列化コンパイラである．OSCAR 自動並列化コンパイラでは，プログラムの基本ブロックやループ，関数呼び出しどうしを並列化する粗粒度タスク並列化，ループイタレーションレベルで並列化を行う中粒度並列化，基本ブロック内のステートメント間の並列化を行う近細粒度並列化の，3 種類の並列化を階層的に行うマルチグレイン並列化を行う [9]．

OSCAR コンパイラはフロントエンド，ミドルパス，バックエンドの 3 つのモジュールから構成される．まず，フロントエンドで入力の C 言語や Fortran で書かれたコードの構文解析や字句解析をし，その結果をコンパイラの中間表現に変換する．次に，ミドルパスで，プログラムを粗粒度タスク並列処理の単位である基本ブロック，繰り返しブロック，および関数呼び出しに分割しこれらをマクロタスク（MT）とする．これら MT 間のコントロールフロー解析とデータ依存解析の結果からマクロフローグラフ（MFG）を生成する．さらに MFG から MT 間のデータ依存とコントロール依存を解析し，各 MT の最早実行可能条件，すなわち MT 間の並列性を解し，MT 間の並列性を図的に表現するマクロタスクグラフ（MTG）を生成する．さらに，キャッシュ最適化やメモリ最適化および各プロセッサへの

コード割当をし，最適化された中間表現を生成する．最後に，バックエンドでは，ミドルパスで生成された中間表現を OSCAR API ディレクティブによって並列化された C 言語や Fortran のプログラムに変換する．

本稿においては，機械学習の学習済みモデルを TVM を用いて C 言語の逐次ソースコードに変換した後，並列化を行うために使用した．TVM と OSCAR を用いた機械学習モデルのコンパイルフローは図 1 に示したとおりである．学習済みモデルを本稿で提案した拡張を施した TVM を用いて C ソースコードに変形し，OSCAR コンパイラを通すことで並列化済み C ソースコードを出力し，NEC の SX-Aurora TSUBASA コンパイラで生成した実行バイナリにより評価した．

3. TVM

TVM（Tensor Virtual Machine）は，オープンソースの深層学習コンパイラおよび深層学習推論ランタイムである．深層学習コンパイラは，学習済みの深層学習モデルを様々なバックエンドに向けて最適化してコード生成する．TVM は複数の深層学習フレームワーク（PyTorch，TensorFlow，Keras など）をサポートしており，各フレームワークで構築された学習モデルを取り込むことができる．また，TVM は複数のバックエンドを持つことで CPU, GPU, DSP などの複数のターゲットハードウェア用の実行コードを生成可能である．TVM では入力としたモデルを，グラフレベルとオペレーターレベルの最適化を行い，ターゲットハードウェア用バックエンドでハードウェアにあわせた効率的なコードを生成する．グラフレベルの最適化では，複数の演算をまとめて中間結果のメモリアクセスを削減するオペレータ融合や，静的に決定できる値を事前に計算して，実行コストを節約する定数畳み込みなどの最適化を行う．オペレータレベルでの最適化では，ループ構造や並列パターンなどの実行詳細を決定するスケジュール変換などの最適化を行う．本稿では，学習済みの機械学習モデルを，OSCAR コンパイラに入力する C ソースコードへ変換するために使用した．

TVM の全体の構成は図 2 のとおりである．

図 2 に示すように，TVM に入力されたモデルは，それぞれのモデルの形式に対応するフロントエンドによって，Computational Graph と呼ばれるグラフレベルの中間表現に変換される．Computational Graph の概要は図 3 の通りである．

Computational Graph は抽象構文木（AST）の形で表され，Python のコードや図 3 左下のようなテキストの形で表される．TVM では，この中間表現に対して，畳み込み演算と活性化関数の計算の融合などのグラフレベルでの最適化を行う．その後，最適化済みのグラフを用いてオペレータレベルでの最適化を行ったうえで，コードを生成する．オペレータレベルの最適化は Tensor Expression と呼ばれる図 4 のような形式で表される中間表現を使って実行される．

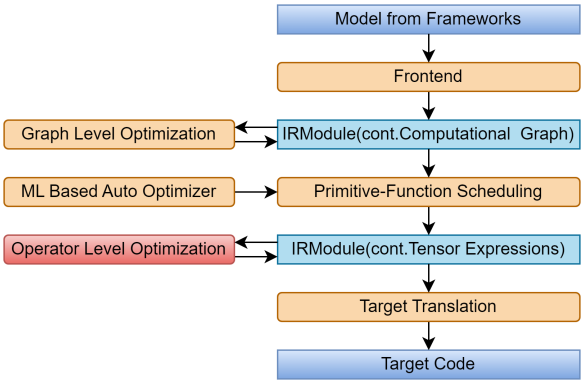

図 2　TVM の構成 [10]

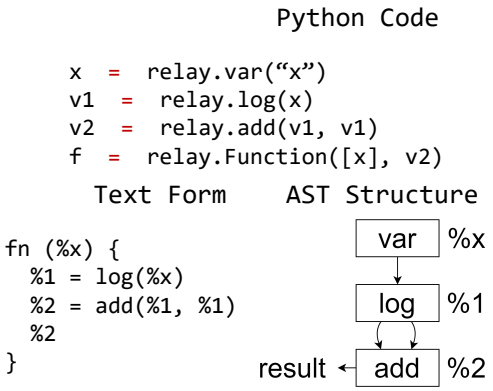

図 3　Computational Graph[11]

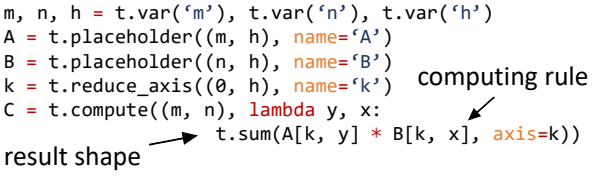

図 4　Tensor Expression[6]

この中間表現は，テンソルの出力形状や演算ルールなどをもとに記述される．オペレータレベルの最適化には，多重ループの複数のループの融合や，ループネストのソートなどの最適化が含まれる．本稿では，この過程に変更を加え，ベクトルアーキテクチャを持つ計算機上での機械学習推論の高速化を図った．

4. 提案する TVM の拡張

本節では本稿で提案する，深層学習推論のベクトルアク

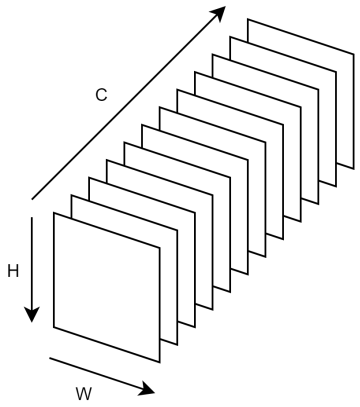

図 5 NCHW レイアウトの概要 [12]

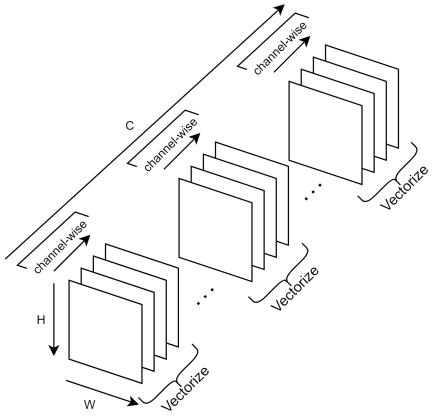

図 6 NCHWc レイアウトの概要

セラレータによる高速化を可能とする TVM の拡張について述べる．

4.1 テンソルレイアウトの変換

一般にループを含むプログラムをベクトル化する場合，メモリアクセスが連続的であることや，ハードウェアのベクトル演算の幅とループ内のデータの幅が一致しているということが重要である．メモリアクセスが連続的であることでデータを連続的に読み込むことができ，メモリバンド幅を最大限に活用できる．また，キャッシュラインの効率的なプリフェッチやキャッシュヒットの増加にもつながり，データの再利用性が高まるため，実行時間の短縮が期待できる．このため，データへのアクセスが連続でかつベクトル化するループ長を十分な長さになるようにテンソルのレイアウトを変更した．

TVM では，バックエンドが C 言語でコード生成する場合，テンソルを NCHWc のレイアウトで扱う．ここで N はバッチサイズ，C はチャネル数，H は高さ，W は幅である．NCHWc のレイアウトは，図 5 のような ONNX や PyTorch などで用いられる NCHW のレイアウトのチャネル内の複数の要素を 1 つの channel-wise としてさらに細かく並べる，図 6 のようなデータ構造となっている．

CNN における畳み込み演算では，出力チャネル（図 5 の C）方向にベクトル化できる [12]．また，内部的な次元を出力特徴マップに追加することで，連続したメモリアクセスが可能となる [13]．本稿では，ベクトル演算における同時演算性能を大きくするために，ベクトル化される最内側のループ長を十分な長さになるよう，channel-wise の大きさを拡大した．channel-wise の大きさを拡大することで，図 6 のように，ベクトル化して同時計算できるテンソルの数を増やし，推論処理の高速化を図った．評価に使用した NEC SX-Aurora TSUBASA のベクトル長が 256 であることから，各演算の出力テンソルにおける channel-wise 方向の大きさが最大 256 となるように変形した．さらに，それに伴いカーネルのレイアウト変形も行った．

4.2 出力コードの形状

実際にテンソルレイアウトの編集をする前の出力 C コードは図 7 のとおりであった．

このコードでは最内側のループ長が 4 となっている．次に，テンソルレイアウトの編集をした後の畳み込み層の出力コードを図 8 に示す．

提案するテンソル変換の拡張を施した TVM による出力コードでは，最内側のループなど，チャネル方向の内側のループ長は 256 となっている．

5. 性能評価

5.1 評価方法

本節で提案するベクトルマルチコア用拡張を適用した TVM の性能評価について説明する．評価には，スキップ接続を導入することで勾配消失問題を軽減し，深いネットワークの構築を可能とした ResNet[14] より，ResNet50 の ImageNet1000 画像分類の学習済みのための ONNX 形式のモデルを用いた．本モデルを拡張前後の TVM を用いて C 言語の逐次ソースコードに変換し，それぞれ OSCAR 自動並列化コンパイラを用いて並列化などの最適化を行った場合と，OSCRA 自動並列化コンパイラを用いない場合と両方のソースコードを評価環境 NEC SX-Aurora TSUBASA のネイティブコンパイラである NCC を用いてコンパイルし，ターゲット向けのバイナリコードを生成した．ターゲットとなる NEC SX-Aurora の最大使用コア数は 8 コアであるため，OSCAR 自動並列化コンパイラを用いた評価においては 1,2,4,8 コアに向けた並列化コードを生成し，その並列性を評価した．評価では画像一枚あたりの処理時間に加えて，ベクトル長を測定した．OSCAR コンパイラを用いた場合については，評価用画像を入力し出力コードを実行した結果となる出力バイナリを，同様のモデルの実行結果の出力と比較して，並列化後も等価なコードが実行さ

```
int32_t ax0_ax1_fused_ax2_fused;
for (ax0_ax1_fused_ax2_fused = 0; ax0_ax1_fused_ax2_fused < 896; ++ax0_ax1_fused_ax2_fused) {
  static float conv2d_NCHWc_global[56];
    {
      int32_t ic_outer;
      for (ic_outer = 0; ic_outer < 64; ++ic_outer) {
        {
          int32_t kh;
          for (kh = 0; kh < 3; ++kh) {
            {
              int32_t kw;
              for (kw = 0; kw < 3; ++kw) {
                {
                  int32_t ic_inner;
                  for (ic_inner = 0; ic_inner < 4; ++ic_inner) {
                    {
                      int32_t ow_c_inner;
                      for (ow_c_inner = 0; ow_c_inner < 14; ++ow_c_inner) {
                        {
                          int32_t oc_block_c;
                          for (oc_block_c = 0; oc_block_c < 4; ++oc_block_c) {
                            conv2d_NCHWc_global[((ow_c_inner * 4) + oc_block_c))] = (conv2d_NCHWc_global
                              [((ow_c_inner * 4) + oc_block_c))] + (data_pad[(((((((ic_outer * 3364) + ((
                              ax0_ax1_fused_ax2_fused % 14) * 232)) + (kh * 116)) + (ow_c_inner * 8)) + (
                              kw * 4)) + ic_inner))] * arg_placeholder1[(((((((ax0_ax1_fused_ax2_fused /
                              14) * 9216) + (ic_outer * 144)) + (kh * 48)) + (kw * 16)) + (ic_inner * 4))
                              + oc_block_c))]));
                          }
                          ...
```

図 7　編集前の畳み込みループ

表 1　評価対象

TVM	Ver.0.8.0[15]
モデル	ResNet50（ImageNet1000 画像分類）[16]
入力画像	224×224×3（cat.png）[17]

表 2　NEC SX-Aurora TSUBASA Vector Engine の構成

Vector Core	8cores
動作周波数（VE）	1.4GHz
メモリ容量	24GB
L1d Cache	32KiB / core
L1i Cache	32KiB /core
L2 Cache	256KiB / core
L3 Cache （LLC）	16MiB
ベクトルレジスタ数	64（プロセッサあたり）
ベクトル長	256 要素

れていることを確認した．一方で NCC のみを用いて実行した場合は，1 コアでは正答となるものの，2 コア以上の並列化を行った場合には正答とならなかった．使用したモデル，TVM，推論に用いた画像などは表 1 の通りである．

5.2　評価環境

ベクトルアクセラレータを搭載した NEC SX-Aurora TSUBASA 上で評価した．評価に用いた NEC SX-Aurora TSUBASA の構成は表 2 の通りである．

尚，評価に用いたモデルは A100-1 で，Type 10C の Vector Element （VE）を 1 機搭載している．VE の動作周波数は 1.4GHz で，6MiB の LLC と 64 個のレジスタがある．また，最大のベクトル長は 256 要素である．評価に使用したコンパイラ NCC のバージョンは 3.5.1 である．最適化は O3 の最適化で，OSCAR コンパイラを使用しない場合は-mparallel のオプションを付け並列化した一方，OSCAR コンパイラを使用して並列化した場合は OpenMP フラグに対応するため，-fopenmp のオプションを付け，NCC の並列化を無効化する-mno-parallel オプションをつけた．

5.3　テンソルレイアウト変更前の評価結果

まず，提案手法適用前の TVM を用いて ResNet50 の学習済みモデルから得られたコードを OSCAR コンパイラを用いて並列化して実行した．評価結果を表 3 に示す．実行時間は 1 コアで約 2 秒あり，8 コア時の速度向上率は 1 コア時に対して 2.3 倍となった．

また，OSCAR コンパイラを用いずに NCC で並列化を行った場合の評価結果は表 4 に示す．NCC で並列化を行った場合には，マルチコア実行時にも実行時間は高速化されなかった．

```
int32_t oh;
  for (oh = 0; oh < 14; ++oh) {
  {
    int32_t ow_outer;
    for (ow_outer = 0; ow_outer < 14; ++ow_outer)
      {
      {
        int32_t kh;
        for (kh = 0; kh < 3; ++kh) {
          {
            int32_t kw;
            for (kw = 0; kw < 3; ++kw) {
              {
                int32_t ic_inner;
                for (ic_inner = 0; ic_inner < 256;
                    ++ic_inner) {
                  {
                    int32_t oc_block_c;
                    for (oc_block_c = 0;
                      oc_block_c < 256; ++
                      oc_block_c) {
                      conv2d_NCHWc_global[(
                        oc_block_c)] = (
                        conv2d_NCHWc_global[(
                        oc_block_c)] + (data_pad
                        [((((((oh * 14848) + (kh *
                         7424)) + (ow_outer * 512)
                        ) + (kw * 256)) + ic_inner
                        ))] * arg_placeholder1
                        [((((((kh * 196608) + (kw *
                         65536)) + (ic_inner *
                        256)) + oc_block_c))]));
                    }
                    ...
```

図 8　編集後の畳み込みループ

表 3　拡張前の TVM 出力の C コードの実行時間

コア数	実行時間 [ms]	速度向上率 (対 1 コア比)
1	2,077.9	—
2	1,489.3	1.4
4	974.7	2.1
8	901.4	2.3

表 4　拡張前の TVM 出力の C コードの実行時間 (NCC のみ)

コア数	実行時間 [ms]
1	2,181.3
2	2,181.3
4	2,181.2
8	2,181.2

5.4　テンソルレイアウト変更後の評価結果

次に，提案するテンソルレイアウトの変更の処理を実装した TVM を用いて評価した．提案手法を実装した TVM から得られたテンソルレイアウト変更適用のコードを OSCAR コンパイラを通さず，そのまま NCC を通して実行した場合（NCC のみ）と，同じコードを OSCAR コンパイラ

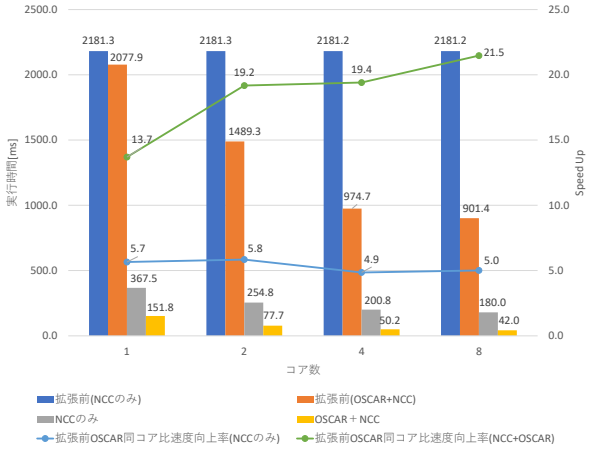

図 9　テンソルレイアウト変更後の実行時間

を用いて並列化したうえで NCC に通してコンパイルした場合（OSCAR+NCC）の評価結果は図 9 に示す．図 9 より，テンソルレイアウトの変更を行う前の OSCAR コンパイラを用いて並列化した場合の実行時間は 1 コアで 2077.9 ミリ秒，8 コアで 901.4 ミリ秒となっているのに対し，テンソルレイアウト変更適用のコードでは，OSCAR コンパイラを用いて並列化した場合には 1 コアで 151.8 ミリ秒，8 コアで 42.0 ミリ秒，NCC のみを用いた場合でも 1 コアで 367.5 秒，8 コアで 180.0 ミリ秒となっており，テンソルのレイアウト変更により，NCC のみの場合と OSCAR コンパイラを通した場合の両方で，シングルコア実行時とマルチコアでの実行時のいずれの場合も実行時間は短縮された．テンソルのレイアウト変更前の OSCAR コンパイラを用いた場合の評価結果と比較すると，シングルコア時は，NCC のみで 5.7 倍，OSCAR コンパイラを用いた場合は 13.7 倍であり，さらに 8 コア時には 21.5 倍とそれぞれ性能向上が得られた．

次に並列化の状況を調べるため，シングルコア実行時に対するマルチコア実行時の高速化率を調べた．この結果を図 10 に示す．テンソルのレイアウト変更前の OSCAR コンパイラ使用時には 1 コア時に対する 8 コア時の高速化率は 2.3 倍であり，テンソル変形後には NCC のみで 2.0 倍，NCC+OSCAR で 3.6 倍となった．テンソルのレイアウト変更前の並列化時の速度向上率は変更後の NCC のみのものよりも大きくなっており，また，テンソルのレイアウトを変更したうえで OSCAR コンパイラを用いた場合は，並列化時の速度向上率が他の 2 つよりも大きくなった．

推論の処理のための総実行命令数は図 11 に示す．テンソルレイアウトを変更する前，NCC で並列化した場合（図 11 拡張前（NCC のみ））には 1 コアで 19.8 億命令，8 コアで 89.2 億命令，OSCAR コンパイラを用いて並列化した場合（図 11 拡張前（NCC+OSCAR））には 1 コアで 18.0 億命令，8 コアでは 42.4 億命令が実行されていたのに対し，テンソルのレイアウトを変更したコードで OSCAR コンパ

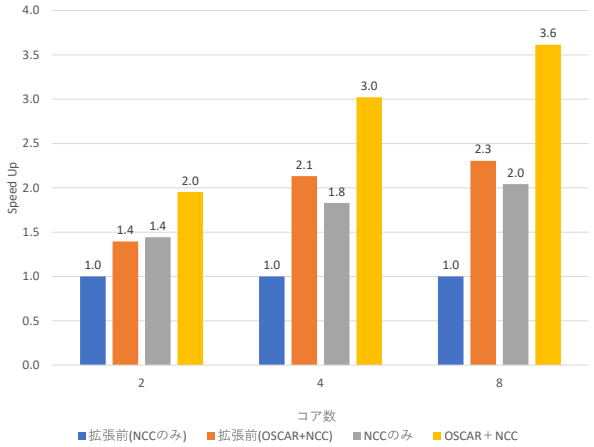

図 10 マルチコア実行時の高速化率

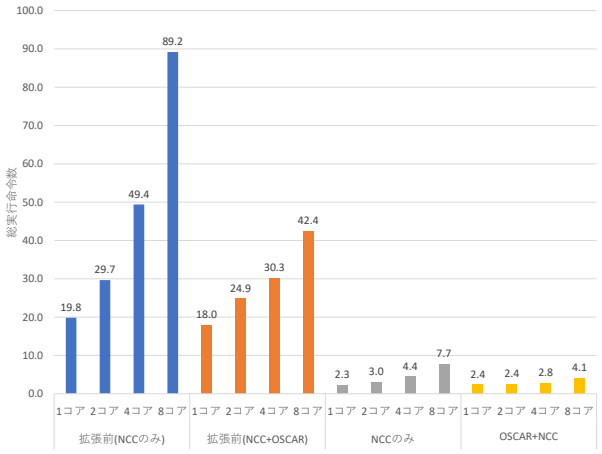

図 11 総実行命令数

イラを用いた場合（図 11 NCC+OSCAR）には，1 コアで 2.4 億命令，8 コアでも 4.1 億命令となり，実行された命令の数が大幅に減少していることがわかる．このように総実行命令数が減少していることは，実行時間の削減の 1 つの要因となっている．また，テンソルのレイアウト変更後の OSCAR 有りの場合（図 11 NCC+OSCAR）と無しの場合（図 11 NCC のみ）を比較すると，1 コアの場合は OSCAR なしで 2.3 億命令，OSCAR 有りの場合は 2.4 億命令なのに対し，8 コアに並列化した場合は，OSCAR なしの場合は 7.7 億命令，OSCAR 有りの場合は 4.1 億命令となっており，OSCAR コンパイラを用いて並列化した場合には命令数の増加を抑えられていることがわかる．テンソルのレイアウト変更によりベクトル命令数やベクトル長を大きくは大きくなり，これにより総実行命令数が削減できた．

テンソルのレイアウト変更前後のベクトル命令の割合を 12 に示す．変更する前は NCC のみ（図 12 拡張前（NCC のみ））で 77.0%，OSCAR コンパイラを用いた場合（図 12 拡張前（NCC+OSCAR））に 83.5%であったベクトル命令の割合は，テンソル変形により，NCC のみ（図 12 NCC のみ）で 97.4%，OSCAR コンパイラを併用した場合（図 12

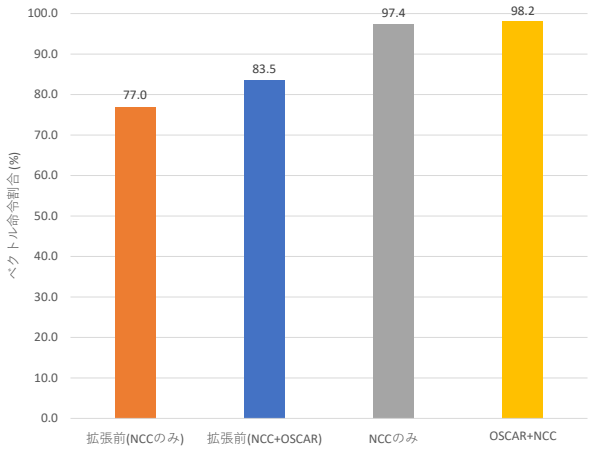

図 12 全命令に占めるベクトル命令の割合

NCC+OSCAR）には 98.2%となり，ベクトル命令の割合が大きくなっていることがわかる．

また，平均のベクトル長を図 13 に示す．テンソルのレイアウト変更前は NCC のみ（図 13 拡張前（NCC のみ））で 14.9，OSCAR コンパイラを用いた場合（図 12 拡張前（NCC+OSCAR））でに 15.3 だったベクトル長は，変更後には，NCC のみ（図 12 NCC のみ）で 166.8，OSCAR ＋ NCC（図 12 NCC+OSCAR）で 160.2 となり，テンソルのレイアウト変更によってベクトル長は 10 倍以上に大きくなっていることがわかる．さらに，ベクトル演算の総実行時間は，レイアウト変更前に OSCAR コンパイラを用いると 255.7 ミリ秒だったのに対し，変更後には 76.5 ミリ秒となっており，ベクトルロード命令の実行時間も 1701.5 ミリ秒から 59.5 ミリ秒に短縮されている．ベクトル長が大きくなることで，ベクトル命令の実行回数が減少していることがわかる．NCC のみの場合と比較してもベクトルロード命令の実行時間は 255.7 ミリ秒から 59.5 ミリ秒に減少しており，メモリへのアクセス回数が最適化により減ったことで実行時間の短縮に繋がった．

以上のように，ベクトル長が大きくなり，かつベクトル命令の割合も大きくなったことで，実行時間の大幅な削減に繋がった．

6. まとめ

本稿では，深層学習コンパイラ TVM を用いたベクトルマルチコア向けコード生成手法を提案した．深層学習の学習済みモデルを深層学習コンパイラ TVM を用いて C ソースコードに変換し，OSCAR コンパイラを用いて並列化などの最適化を行ったうえで実行バイナリを生成した．学習モデルを C ソースコードに変換する過程で，ベクトル化の効果を意識して，最内側のループ長が大きくなるようにテンソルレイアウトを変形した．この結果，テンソルレイアウトの変形前の同コア比での推論処理の実行時間は，1 コアでは 13.5 倍，8 コアでは 21.5 倍にそれぞれ高速化され

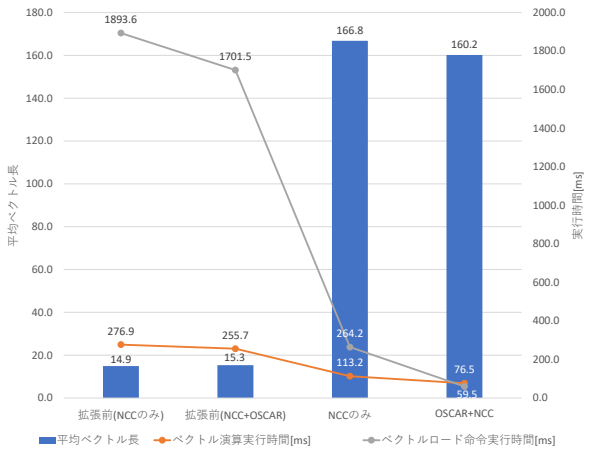

図 13 平均ベクトル長とベクトル命令実行時間

た．また，テンソルレイアウトを変形した C ソースコードを用いた場合，OSCAR コンパイラを通したときと NCC のみでコンパイルしたときとを比較すると，1 コア実行時と 8 コア実行時との速度比は 2.0 から 3.6 まで大きくなった．以上から，深層学習モデルを，TVM を用いて C ソースコードに変換し，これを OSCAR コンパイラで解析することで，自動で並列化を行うことが確認できた．さらに，TVM が生成する C ソースコードのテンソルレイアウトが，畳み込みの最内側のループ長が大きくなるように変換することで，ベクトルプロセッサの実行効率が向上し，より高速な実行が可能となることを確認した．

謝辞 本研究の成果の一部は JST【ムーンショット型研究開発事業】【JPMJMS2031】の支援を受けたものです．

参考文献

[1] Hirano, T., Yamamoto, H., Iizuka, S., Muto, K., Goto, T., Wake, T., Mikami, H., Takamura, M., Kimura, K. and Kasahara, H.: Evaluation of Automatic Power Reduction with OSCAR Compiler on Intel Haswell and ARM Cortex-A9 Multicores, *Proceedings of the 27th International Workshop on Languages and Compilers for Parallel Computing (LCPC)*, LCPC (2014).

[2] Kasahara, H., Kimura, K., Kitamura, T., Mikami, H., Morita, K., Fujita, K., Yamamoto, K. and Kawasumi, T.: OSCAR Parallelizing and Power Reducing Compiler and API for Heterogeneous Multicores: (Invited Paper), *2021 IEEE/ACM Programming Environments for Heterogeneous Computing (PEHC)*, pp. 10–19 (online), DOI: 10.1109/PEHC54839.2021.00007 (2021).

[3] Abadi, M., Agarwal, A., Barham, P., Brevdo, E., Chen, Z., Citro, C., Corrado, G. S., Davis, A., Dean, J., Devin, M. et al.: TensorFlow: Large-Scale Machine Learning on Heterogeneous Distributed Systems, *arXiv preprint arXiv:1603.04467* (2016).

[4] Paszke, A., Gross, S., Massa, F., Lerer, A., Bradbury, J., Chanan, G., Killeen, T., Lin, Z., Gimelshein, N., Antiga, L. et al.: PyTorch: An Imperative Style, High-Performance Deep Learning Library, *arXiv preprint arXiv:1912.01703* (2019).

[5] Bai, J., L. F. Z. K. e. a.: ONNX: Open Neural Network Exchange, GitHub (online).

[6] Chen, T., Moreau, T., Jiang, Z., Zheng, L., Yan, E., Cowan, M., Shen, H., Wang, L., Hu, Y., Ceze, L., Guestrin, C. and Krishnamurthy, A.: TVM: An Automated End-to-End Optimizing Compiler for Deep Learning, *arXiv preprint arXiv:1802.04799* (2018).

[7] NEC Corporation: NEC SX-Aurora TSUBASA Architecture - NEC Vector Engine Processor, available from (`https://www.nec.com/en/global/solutions/hpc/sx/architecture.html`) (accessed 2023-06-25).

[8] Kimura, K., Mase, M., Mikami, H., Miyamoto, T., Shirako, J. and Kasahara, H.: OSCAR API for Real-Time Low-Power Multicores and Its Performance on Multicores and SMP Servers, *Languages and Compilers for Parallel Computing* (Gao, G. R., Pollock, L. L., Cavazos, J. and Li, X., eds.) (2010).

[9] Kasahara, H., Honda, H., Mogi, A., Ogura, A., Fujiwara, K. and Narita, S.: A Multi-grain Parallelizing Compilation Scheme for OSCAR (Optimzally Scheduled Advanced Multiprocessor), *Proc. 4th Intl. Workshop on LCPC*, pp. 283–297 (1991).

[10] TVM: Design and Architecture (online), available from (`https://tvm.apache.org/docs/arch/index.html`) (accessed 2023-06-25).

[11] TVM: Introduction to Relay IR (online), available from (`https://tvm.apache.org/docs/arch/relay_intro.html`) (accessed 2023-06-25).

[12] 大髙凌聖，小池穂乃花，磯野立成，川角冬馬，北村俊明，見神広紀，納富 昭，木村貞弘，木村啓二，笠原博徳：各コアがローカルメモリを持つ組み込みベクトルマルチコアでの畳み込み層演算の評価，*2023-EMB-62(32)*, pp. 1–6 (2023).

[13] Das, D., Avancha, S., Mudigere, D., Vaidynathan, K., Sridharan, S., Kalamkar, D., Kaul, B. and Dubey, P.: Distributed Deep Learning Using Synchronous Stochastic Gradient Descent, *arXiv preprint arXiv:1602.06709* (2016).

[14] He, K., Zhang, X., Ren, S. and Sun, J.: Deep Residual Learning for Image Recognition, *CoRR*, Vol. abs/1512.03 (2016).

[15] APACHE: TVM: Open deep learning compiler stack for cpu, gpu and specialized accelerators, GitHub (online), available from (`https://github.com/apache/tvm/tree/v0.8`) (accessed 2023-06-25).

[16] ONNX: ONNX Models: ResNet model, GitHub (online), available from (`https://github.com/onnx/models/tree/main/vision/classification/resnet/model/resnet50-v2-7.onnx`) (accessed 2023-06-25).

[17] DMLC: MXNet.js: Javascript Package for Deep Learning in Browser, GitHub (online), available from (`https://github.com/dmlc/mxnet.js/blob/master/data`) (accessed 2023-06-25).

正誤表

下記の箇所に誤りがございました．お詫びして訂正いたします．

訂正箇所	誤	正
1 ページ 14 行目	1 コア実行時に 14.0 倍，8 コア時に 27.1 倍の速度向上をそれぞれ得た．	1 コア実行時に 13.7 倍，8 コア時に 21.5 倍の速度向上をそれぞれ得た．