

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

大高 凌聖^{1,a)} 小池 穂乃花¹ 磯野 立成¹ 川角 冬馬¹ 北村 俊明¹ 見神 広紀² 納富 昭²
木村 貞弘³ 木村 啓二¹ 笠原 博徳¹

概要: IoT デバイスの普及に伴い、これらの組み込みシステム上でも深層学習利用に対する要求が増えてきた。例えば、自動運転での路面画像等の周辺環境情報の認識処理による障害物判定等はその性質上走行中の自動車内部で行う必要がある。しかしながら、組み込みシステムでは PC のような大規模かつ消費電力の大きいハードウェア資源を利用することは困難である。そのため、プログラムの並列性を積極的に利用し、低動作周波数の演算器やプロセッサコアをベクトル処理及び並列処理することにより電力効率の高いシステムを構成することが重要となる。本稿では、画像認識処理で広く用いられている畳み込みニューラルネットワーク (CNN) の主要処理である畳み込み層と活性化関数に対して、手動でベクトル化とそれらに伴う各種最適化を適用した。さらに、各コアがローカルメモリとベクトルアクセラレータを持つマルチコアを想定し、マルチコア並列化とそれに伴うデータのローカルメモリ配置とデータ転送挿入を行った。これらのプログラムを、FPGA 評価ボード上に構築した OSCAR ベクトルマルチコア上で評価した。評価の結果、1PE 時のベクトル化のみを行ったものから最適化を行った結果 3.36 倍の速度向上を得た。さらに、本ベクトル最適化を行ったものを 4PE で実行した結果の 1PE 時と比較して 2.94 倍の速度向上を得た。

1. はじめに

IoT デバイスをはじめとする組み込みシステムは既に社会の隅々まで配置・利用されている必要不可欠な社会インフラとなっている。そのため組み込みシステムに対する要求も高度化し、深層学習を利用したサービスも増えてきている。例えば、自動車の運転補助やさらに自動運転システムではカメラ画像等の周辺環境を認識 [1] し車体の制御を行う。このとき、応答性やそれに伴う運転の安全性等の理由から、認識処理は車内搭載のシステムで行う必要がある。

しかしながら、これらの組み込み機器は、その設置スペースと電源供給能力が制限されている場合が多い。そのため、サービスが要求する性能を満たすために潤沢なハードウェア資源を投入し高い発熱を伴う消費電力の高いシステムを導入するわけにはいかない。すなわち、深層学習に対する要求の高まりと共に電力効率の高い組み込みシステムに対する要求もさらに高まっている。

電力効率の高いコンピュータシステムの構成方法に関しては、プログラムから並列性を抽出した上で、複数の演算器やプロセッサコアを適切な周波数と電圧で駆動することが有効である [5]。例えば、高動作周波数の単一プロセッサコアによる逐次処理に対し、低動作周波数・低電圧の複数コア駆動の並列処理により、同一性能を低消費エネルギーで達成可能である [6]。このような並列処理による低消費電力なコンピュータシステムの実現を自動並列化コンパイラとマルチコアアーキテクチャ協調により達成する目的で、筆者等は各コアが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のためのベクトル/マルチコア並列化・メモリ最適化・電力最適化を行う OSCAR コンパイラの開発を行っている [7]。

本稿では、画像認識処理で広く用いられている畳み込みニューラルネットワーク (CNN) の主要処理である畳み込み層と活性化関数を対象に、これらを OSCAR ベクトルマルチコア上で処理する方法を検討した結果について報告する。より具体的には、コンボリューション層で処理される入出力特徴量マップテンソル及び重みテンソルの形状とそれらの演算処理の特性を踏まえてベクトル化の方針を決定した。次に、ベクトルレジスタ上のデータローカリティを最大限活かすためのループアンローリングとレジスタブ

¹ 早稲田大学

Waseda University.

² オスカーテクノロジー

Oscar Technology Corporation

³ エヌエスアイテクス

NSITEXE

a) suan@kasahara.cs.waseda.ac.jp

ロッキングを導入することで、ベクトルパイプラインの利用効率を向上しベクトルマルチコア 1PE 実行の最適化を施した。さらにこのような 1PE ベクトル最適化を前提として、マルチコア並列化を適用した。このとき、各コアのローカルメモリ上にデータを配置し、これらローカルメモリとチップ外部の主記憶（集中共有メモリ）間の DMA によるデータ転送を演算処理とオーバーラップ実行するコードも挿入した。これらのベクトル化・マルチコア並列化による最適化を施したプログラムを、FPGA 評価ボード上に構築した OSCAR ベクトルマルチコア上で評価実験を行った。

本稿の構成を以下に示す：2 節では今回使用する組み込みベクトルマルチコアである OSCAR ベクトルマルチコアアーキテクチャとそのベクトルアクセラレータアーキテクチャについて、3 節では評価対象である CNN の主要演算である畳み込みと活性化関数について、4 節では今回行ったベクトル化及び並列化と最適化について、5 節では性能評価についてそれぞれ述べ、最後に 6 節でまとめる。

2. OSCAR ベクトルマルチコアアーキテクチャ

2.1 OSCAR ベクトルマルチコアアーキテクチャ

OSCAR ベクトルマルチコアアーキテクチャは OSCAR マルチコアアーキテクチャ [2] をベースとしている（図 1）。一つのチップはオンチップ集中共有メモリ（On-chip CSM）と複数のプロセッサコア（プロセッサエレメント：PE）を搭載し、各 PE は CPU、ベクトルアクセラレータ（VA）、ローカルデータメモリ（LDM）、分散共有メモリ（DSM）、データ転送ユニット（DTU）を持つ。また、チップ外部には主記憶としてオフチップ集中共有メモリ（Off-chip CSM）が接続されている。

LDM は自 PE 内の CPU、VA、DTU のみがアクセス可能な高速なメモリである。DSM は各 PE からアクセスが可能なメモリであり、主にタスク間のデータ転送や同期フラグなどのデータが保存される。CSM は LDM や DSM に比べアクセス時間が長いメモリだが、容量が大きく、すべてのプログラムやデータが格納される。VA はベクトル演算器を搭載したアクセラレータであり、各 PE 内の CPU によって起動される。また、VA は LDM 及び DSM に対してのみアクセスが可能であり、CSM にアクセスすることは不可能である。CPU、VA、DTU はそれぞれ非同期に動作が可能である。

2.2 ベクトルアクセラレータの構造

OSCAR ベクトルマルチコアアーキテクチャが持つ VA は、富士通の VPP シリーズで開発されてきたベクトルプロセッサ [3] をベースにしたアクセラレータである。

VA の構造を図 2 に示す。VA にはベクトル演算器とスカラー演算器が搭載されており、使用するレジスタとしてベ

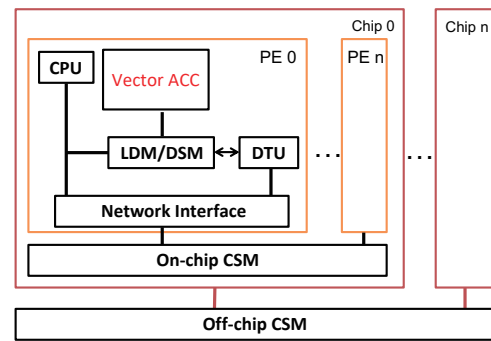


図 1 OSCAR ベクトルマルチコアアーキテクチャ

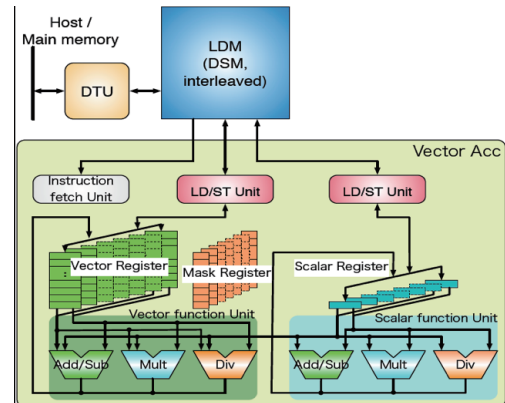


図 2 ベクトルアクセラレータ

クトルレジスタ（VR）、マスクレジスタ（MR）、スカラーレジスタ（SR）が搭載されている。VR は 256bytes のサイズを持ち、単精度浮動小数点データであれば 64 要素格納可能である。また、2.1 節に示した通り CPU とは独立に動作が可能であり、簡易的なループや条件分岐などは VA 単体で動作が可能である。

ベクトル命令に利用するベクトルの長さはベクトルレジスタのサイズの範囲内で自由に指定が可能である。また、MR を利用することによりマスク演算をすることが可能になり、簡単な分岐条件を伴う演算もベクトル化可能である。

本ベクトルアクセラレータのコード生成を行うために LLVM[9] を拡張した [7]。拡張した LLVM はベクトル命令に対応する組み込み関数を解釈し、入力となる C/C++ プログラムからベクトル命令を含むアセンブリを生成する。

3. 畳み込みニューラルネットワークにおける演算

本節では本稿が対象とする畳み込みニューラルネットワーク（CNN）の主要処理である畳み込み計算と活性化関数について述べる。

3.1 CNN における畳み込み演算

畳み込み演算は図 3 に示すように、入力データにフィルタ（重み）を掛け合わせていくことにより入力データから特徴量を抽出してそれを出力データとする。CNN にお

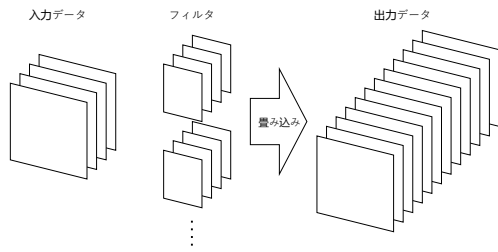


図3 畳み込み演算とその入出力データの概要

る畳み込み演算は、入出力データ共に複数チャンネルの2次元データから構成される。

畳み込み演算のデータフローに着目すると、例えば 3×3 のフィルタによる畳み込みを行う場合、入力データの着目要素を中心とした 3×3 要素を全チャンネルに渡って畳み込み演算を行い1出力チャンネル分の1要素の値を算出する。これを全出力チャンネル用のフィルタに対して行い、出力データを得る。出力データの観点からは、入力データ1チャンネル分の要素に対する各出力チャンネル用フィルタの畳み込み演算により全出力チャンネルの中間結果を算出し、これを全入力チャンネルに対して行い出力データに足し込むことで最終的な出力データを得る。

すなわち、1入力チャンネルの着目要素に対する畳み込み演算、及び1出力チャンネル要素の算出はリダクション演算を要求する。一方、1入力チャンネルの着目要素から全出力チャンネルの中間結果の算出は入力要素のスカラ値と出力チャンネル用フィルタ値のベクトルデータの積という、出力チャンネル方向でベクトル演算化が可能である。

3.2 活性化関数

畳み込み層の演算結果を入力として、これを次の層へどのようにデータを伝播させるかを活性化関数により調節する。例として、ReLUでは畳み込み層の出力データの値が負なら0を、正ならそのままの結果を出力する。活性化関数の処理は各入力データ間で依存が無く、本処理も出力データのチャンネル方向でベクトル化可能である。

4. ベクトル化及び並列化と最適化

本節で畳み込み層及び活性化関数の性能評価にあたり、OSCARベクトルマルチコアアーキテクチャ用に施した並列化・ベクトル化及び最適化について述べる。

4.1 ベクトル化の方針

CNNの畳み込み層で使用するデータは、入力データ、出力データ、フィルタの3種類がある。入力データ、出力データは画像データもしくはその特徴量を示すデータとなる。フィルタは入力データからある特徴を抽出するために決められたデータが格納される。

本稿では、CNNの出力データのチャンネル方向（図4の

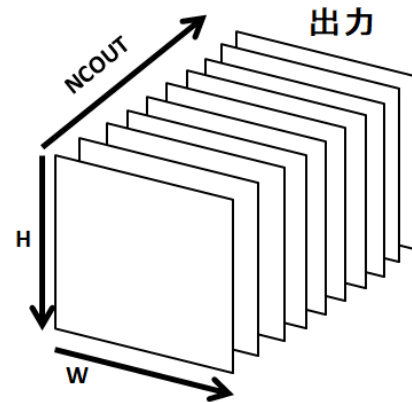


図4 畳み込み層の出力データの形状

NCOUTの方向)でベクトル化する。出力チャンネル方向は3.1節で述べたようにベクトル化可能である。またCNNでは出力チャンネルが比較的長くなる傾向にあり、例えば代表的なCNNの一つであるResNetでは、初期の入力データは $224 \times 224 \times 1$ でチャンネル方向は1だが、その出力は $112 \times 112 \times 64$ となり、チャンネル方向の大きさが64となる[8]。さらに処理が進むと出力データは $7 \times 7 \times 512$ になり、チャンネル方向の大きさは512となる。一般にベクトル化はベクトル長が長くなるほどベクトルパイプラインの効率が向上する。そのため、このような出力データのチャンネル方向でベクトル化することが効率的である。

以上の方針でベクトル化を行ったときの擬似コードを図5に示す。図中、`_pt_vld.f`、`_pt_vmuls.f`、`_pt_vadd.f`はそれぞれベクトルロード、スカラ・ベクトル積、及びベクトル加算をそれぞれ表す。VRnはベクトルレジスタである。また、INaddress、OUTaddress、Waddressはそれぞれ入力要素、出力要素、フィルタ要素のアドレスを表す。本コードでは $5 \times 5 \times 8$ チャンネルに対して 3×3 のフィルタによる畳み込み演算を行っている。出力チャンネルに対してはベクトル長でストリップマイニングしているが、簡単のためストリップマイニング後の外側ループは省略されている。

4.2 ループアンローリングとレジスタブロッキング

3.1節で述べたように、畳み込み演算の出力チャンネル方向のベクトル化は、入力データ1要素とフィルタのベクトルデータのスカラ・ベクトル積により出力チャンネルの中間結果を算出し、これを全入力チャンネルに対して繰り返し出力チャンネルに足し込むことで出力チャンネルの最終結果を得る。すなわち、一度ベクトルレジスタに確保したフィルタの値及び出力チャンネルの中間結果には再利用性がある。そこでオリジナルのベクトル化した畳み込みプログラムをベクトル長でレジスタブロッキングした上で、オリジナルの各入力要素の演算に必要な値をベクトルレジスタが利用可能な範囲でループアンローリングした。これによりベクト

```
for(i = 0; i < 5; i++) {
  for(j = 0; j < 5; j++) {
    VR0 = _pt_vld_f(OUTAddress(i, j, 0));
    for(m = 0; m < 3; m++) {
      for(n = 0; n < 3; n++) {
        for(o = 0; o < 8; o++) {
          VR1 = _pt_vld_f(Waddress(o, m, n, 0));
          VR2 = _pt_vmuls_f(
            *INAddress((i + m - 1), (j + n - 1), (o)),
            VR1);
          VR0 = _pt_vadd_f(VR0, VR2);
        }
      }
    }
  }
}
```

図 5 ベクトル化畳み込み層プログラムの疑似コード

```
VR7 = _pt_vld_f(OUTAddress(0, 0, 0));
VR8 = _pt_vld_f(OUTAddress(0, 1, 0));
~
for(i = 0; i < 8; i++) {
  in_u1_u1 = *INAddress(-1, -1, i);
  in_u1_0 = *INAddress(-1, 0, i);
  in_u1_1 = *INAddress(-1, 1, i);
  ~
  VRD = _pt_vld_f(Waddress(i, 0, 0, 0));
  VRE = _pt_vld_f(Waddress(i, 0, 1, 0));
  ~
  VRC = _pt_vmuls_f(in_u1_u1, VRD);
  VR7 = _pt_vadd_f(VR7, VRC);
  VRC = _pt_vmuls_f(in_u1_0, VRE);
  VR7 = _pt_vadd_f(VR7, VRC);
  ~
  VRC = _pt_vmuls_f(in_u1_1, VRD);
  VR8 = _pt_vadd_f(VR8, VRC);
  VRC = _pt_vmuls_f(in_u1_1, VRE);
  VR8 = _pt_vadd_f(VR8, VRC);
  ~
}
```

図 6 レジスタブロッキング・ループアンローリング後の疑似コード

ルレジスタの再利用性が高まり不要なベクトルロード・ストア命令が削除でき、またループ繰り返しの制御する分岐命令も削除され、結果としてベクトルパイプラインの利用効率が高まる。

図 5 のコードに対してレジスタブロッキング及びループアンローリングを適用した疑似コードを図 6 に示す。図中のループは入力チャネルに対するループであり、1 入力チャネルの各要素に対する処理をループボディで行っている。例えばベクトルレジスタ VRD, VRE はフィルタ値を、ベクトルレジスタ VR7, VR8 は出力各要素の出力チャネル方向の中間結果をそれぞれ保持し、VRD は VR7, VR8 といった複数の出力データの演算に再ロード無しで利用されている。

4.3 データ転送方式

2.1 節で述べたように、VA は LDM 及び DSM のみに

アクセスが可能である。そのため、CSM から計算に必要なデータを LDM または DSM に転送する必要がある。LDM/DSM は一つの畳み込み層に必要なデータを全て格納可能な容量を持っているとは限らず、その時はデータを分割して必要なデータを都度 LDM/DSM にロード・ストアして演算処理を進めなければならない。この際、必要なデータが LDM/DSM に到着するまで VA では計算が行えずデータ転送時間がオーバーヘッドとなってしまう。そこで、ダブルバッファリングを利用して、LDM/DSM 上に分割したデータを 2 セット確保できるようにし、1 セットの演算処理を行っている間、PE 内部の CPU と DTU を利用してもう 1 セットのデータ転送を処理できるようにした。

5. 性能評価

本節では 4 節で示したベクトル化及び並列化、最適化を行ったプログラムを利用して OSCAR ベクトルマルチコアアーキテクチャ上で実行時間の測定を行った結果について述べる。

5.1 評価環境

本評価では、FPGA 評価ボード上に OSCAR ベクトルマルチコアアーキテクチャを実装し使用した。使用した評価ボードは Intel 社の Arria10[4] である。本評価で用いたマルチコアは PE を 4 つ搭載し、各 PE 中の CPU コアとして動作周波数 50MHz で駆動する NIOS II を使用した。各 CPU は 32KB の命令キャッシュを持つが、データキャッシュはキャッシュコヒーレンシの問題を回避するため持たない。本評価では CPU をほぼ使用しないのでデータキャッシュを持たないことによる性能上の影響は無視できる。また、各 PE は動作周波数 50MHz で駆動するベクトルアクセラレータを持つ。各 PE に搭載されているローカルメモリの容量は 64KB である。

NIOS II 用のコンパイラには gcc 5.3.0 を用いた。また、2.2 節で述べたベクトルアクセラレータ用バックエンドコンパイラには Clang/LLVM 3.2 をベースに拡張したものを利用した。最適化オプションには -O1 を用いた。

5.2 評価プログラム

評価には、単純なベクトル化のみ行ったプログラム (図 7)、ループインタチェンジをしてフィルタの 1 辺をアンローリングしたプログラム (図 8、フィルタアンロール 1)、フィルタの両辺共にアンロールしてフィルタのループを削除したプログラム (図 9、フィルタアンロール 2)、及び 4 節で述べたベクトル化・最適化を行ったプログラム (図 10、アンロール・レジスタブロッキング) を用いた。

畳み込み層で使用される入力データ、出力データ、重さと、活性化関数で使用するスケールの重さ、スケールバイアス、バイアスの大きさを表 1 に示す。また、使用した

```
for(n = 0; n < H; n++) {
    for(m = 0; m < W; m++) {
        VR0 = _pt_vld_f(OUTAddress(n, m, 0));
        for(p = 0; p < 3; p++) {
            for(i = 0; i < 3; i++) {
                for(j = 0; j < NCIN; j++) {
                    VR1 = _pt_vld_f(Waddress(j, p, i, 0));
                    VR2 = _pt_vmuls_f(
                        *INAddress((n + p - 1), (m + i - 1), (j)),
                        VR1);
                    VR0 = _pt_vadd_f(VR0, VR2);
                }
            }
        }
        _pt_vst_f(OUTAddress(n, m, 0), VR0);
    }
}
```

図 7 ベクトル化のみのコード

```
for(n = 0; n < H; n++) {
    for(j = 0; j < NCIN; j++) {
        for(p = 0; p < 3; p++) {
            for(m = 0; m < W; m++) {
                in_m_0 = *INAddress((n + p - 1), (m - 1), j);
                in_m_1 = *INAddress((n + p - 1), m, j);
                in_m_2 = *INAddress((n + p - 1), (m + 1), j);
                for(i = 0; i < NCOU; i += VLEN) {
                    __pvf VR1 = _pt_vld_f(Waddress(j, p, 0, i));
                    __pvf VR2 = _pt_vld_f(Waddress(j, p, 1, i));
                    __pvf VR3 = _pt_vld_f(Waddress(j, p, 2, i));
                    __pvf VR4 = _pt_vld_f(OUTAddress(n, m, i));
                    VR4 = _pt_vadd_f(VR4, _pt_vmuls_f(in_m_0, VR1));
                    VR4 = _pt_vadd_f(VR4, _pt_vmuls_f(in_m_1, VR2));
                    VR4 = _pt_vadd_f(VR4, _pt_vmuls_f(in_m_2, VR3));
                    _pt_vst_f(OUTAddress(n, m, i), VR4);
                }
            }
        }
    }
}
```

図 8 フィルタの 1 辺をアンロールしたコード (フィルタアンロール 1)

ベクトルアクセラレータの実装の都合上、全て単精度浮動小数点で演算した。

使用するデータに関しては、全て CSM 上に用意されているものとし、LDM へのデータ転送時間は実行時間の測定に含まれるものとする。また、ベクトル化方針については 4.1 節の通り、出力チャネルの 128 の方向でベクトル化を行った。複数 PE による並列化は、入出力データを縦横で分割することにより行った。

5.3 評価結果

OSCAR ベクトルマルチコアアーキテクチャ上で実行した結果を図 11 に示す。縦軸は 1PE のベクトル化のみを行ったものに対する速度向上率を表している。

1PE で各種最適化を行った場合、ベクトル化のみの状態

```
for(n = 0; n < H; n++) {
    for(j = 0; j < NCIN; j++) {
        for(m = 0; m < W; m++) {
            in_0_m_0 = *INAddress((n - 1), (m - 1), j);
            in_0_m_1 = *INAddress((n - 1), m, j);
            in_0_m_2 = *INAddress((n - 1), (m + 1), j);
            in_1_m_0 = *INAddress(n, (m - 1), j);
            in_1_m_1 = *INAddress(n, m, j);
            in_1_m_2 = *INAddress(n, (m + 1), j);
            in_2_m_0 = *INAddress((n + 1), (m - 1), j);
            in_2_m_1 = *INAddress((n + 1), m, j);
            in_2_m_2 = *INAddress((n + 1), (m + 1), j);
            for(i = 0; i < NCOU; i += VLEN) {
                __pvf VR1 = _pt_vld_f(Waddress(j, 0, 0, i));
                __pvf VR2 = _pt_vld_f(Waddress(j, 0, 1, i));
                __pvf VR3 = _pt_vld_f(Waddress(j, 0, 2, i));
                __pvf VR4 = _pt_vld_f(Waddress(j, 1, 0, i));
                __pvf VR5 = _pt_vld_f(Waddress(j, 1, 1, i));
                __pvf VR6 = _pt_vld_f(Waddress(j, 1, 2, i));
                __pvf VR7 = _pt_vld_f(Waddress(j, 2, 0, i));
                __pvf VR8 = _pt_vld_f(Waddress(j, 2, 1, i));
                __pvf VR9 = _pt_vld_f(Waddress(j, 2, 2, i));
                __pvf VR10 = _pt_vld_f(OUTAddress(n, m, i));
                VR10 = _pt_vadd_f(VR10, _pt_vmuls_f(in_0_m_0, VR1));
                VR10 = _pt_vadd_f(VR10, _pt_vmuls_f(in_0_m_1, VR2));
                VR10 = _pt_vadd_f(VR10, _pt_vmuls_f(in_0_m_2, VR3));
                VR10 = _pt_vadd_f(VR10, _pt_vmuls_f(in_1_m_0, VR4));
                VR10 = _pt_vadd_f(VR10, _pt_vmuls_f(in_1_m_1, VR5));
                VR10 = _pt_vadd_f(VR10, _pt_vmuls_f(in_1_m_2, VR6));
                VR10 = _pt_vadd_f(VR10, _pt_vmuls_f(in_2_m_0, VR7));
                VR10 = _pt_vadd_f(VR10, _pt_vmuls_f(in_2_m_1, VR8));
                VR10 = _pt_vadd_f(VR10, _pt_vmuls_f(in_2_m_2, VR9));
                _pt_vst_f(OUTAddress(n, m, i), VR10);
            }
        }
    }
}
```

図 9 フィルタループを全てアンロールしたコード (フィルタアンロール 2)

表 1 データサイズ

	次元数	サイズ
入力データ	3	14 × 14 × 128
出力データ	3	14 × 14 × 128
重さ	4	128 × 3 × 3 × 128
スケールの重さ	1	128
スケールバイアス	1	128
バイアス	1	128

と比べ、フィルタアンロール 1 で 1.36 倍、フィルタアンロール 2 で 2.06 倍、ループアンロール・レジスタブロッキングで 3.37 倍の性能向上となり、ループアンローリングやレジスタブロッキングによるベクトルレジスタの有効利用とそれに伴うベクトルパイプラインの利用効率向上により性能向上を得ることが出来た。

さらに複数 PE 利用により、ループアンローリング・ブロッキングした 1PE に対して、その 2PE で 1.76 倍 (ベクトル化のみ 1PE に対して 5.92 倍)、4PE で 2.94 倍 (ベク


```

VR7 = _pt_vld_f(OUTAddress(0, 0, 0));
VR8 = _pt_vld_f(OUTAddress(0, 1, 0));
VR9 = _pt_vld_f(OUTAddress(0, 2, 0));
VRA = _pt_vld_f(OUTAddress(0, 3, 0));
~
for(i = 0; i < 8; i++) {
    in_u1_u1 = *INAddress(-1, -1, i);
    in_u1_0 = *INAddress(-1, 0, i);
    in_u1_1 = *INAddress(-1, 1, i);
    in_u1_2 = *INAddress(-1, 2, i);
    in_u1_3 = *INAddress(-1, 3, i);
    ~
    VRD = _pt_vld_f(Waddress(i, 0, 0, 0));
    VRE = _pt_vld_f(Waddress(i, 0, 1, 0));
    VRF = _pt_vld_f(Waddress(i, 0, 2, 0));
    VR1 = _pt_vld_f(Waddress(i, 1, 0, 0));
    ~
    VRC = _pt_vmuls_f(in_u1_u1, VRD);
    VR7 = _pt_vadd_f(VR7, VRC);
    VRC = _pt_vmuls_f(in_u1_0, VRE);
    VR7 = _pt_vadd_f(VR7, VRC);
    VRC = _pt_vmuls_f(in_u1_1, VRF);
    VR7 = _pt_vadd_f(VR7, VRC);
    VRC = _pt_vmuls_f(in_u1_2, VR1);
    VR7 = _pt_vadd_f(VR7, VRC);
    VRC = _pt_vmuls_f(in_u1_3, VR1);
    VR7 = _pt_vadd_f(VR7, VRC);
    ~
}

```

図 10 アンローリング・レジスタブロッキングしたコード

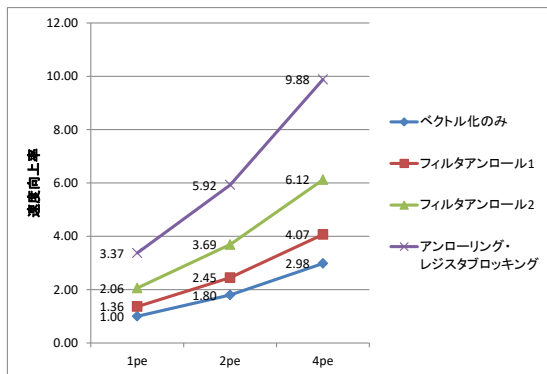


図 11 ベクトル化・並列化による速度向上率

トル化のみ 1PE に対して 9.88 倍) とスケーラブルな性能向上を得ることが出来た。

6. まとめ

本稿では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に向け、CNN の畳み込み層と活性化関数のプログラムを並列化とベクトル化、最適化を行い組み込み向けシステム上で実行し、FPGA 上に構築した OSCAR ベクトルマルチコア上で評価した。評価の結果、単一 PE で実行した場合、レジスタブロッキングやループアンローリ

ングにより 3.36 倍の速度向上を得た。1PE 用に最適化したプログラムをさらに複数 PE による並列化で 2.94 倍の速度向上が得られ、オリジナルの単一 PE プログラムに対して 9.88 倍の速度向上となった。

本稿では手動で実施したベクトルチューニングやローカルメモリ利用・データ転送挿入のツールやコンパイラによる自動化が今後の課題である。

謝辞 本研究の成果の一部は国立研究開発法人新エネルギー・産業技術総合開発機構 (NEDO) の委託業務 JPNP16007 の結果得られたものです。

参考文献

- [1] J. Redmon, S. Divvala, R. Girshick and A. Farhadi, "You Only Look Once: Unified, Real-Time Object Detection," in 2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR), Las Vegas, NV, USA, 2016 pp. 779-788. doi: 10.1109/CVPR.2016.91
- [2] Keiji Kimura, Yasutaka Wada, Hirofumi Nakano, Takeshi Kodaka, Jun Shirako, Kazuhisa Ishizaka, and Hironori Kasahara. Multigrain parallel processing on compiler cooperative chip multiprocessor. In 9th Annual Workshop on Interaction between Compilers and Computer Architectures (INTERACT' 05), pp. 11-20. IEEE, 2005.
- [3] Kenichi Miura, Moriyuki Takamura, Yoshinori Sakamoto, and Shin Okada. Overview of the fu-jitsu vpp500 supercomputer. In Digest of Papers. Compeon Spring, pp. 128-130. IEEE, 1993.
- [4] intel.co.jp: インテル® Arria® 10 FPGA および SoC FPGA
- [5] Jun Shirako, Munehiro Yoshida, Naoto Oshiyama, Yasutaka Wada, Hirofumi Nakano, Hiroaki Shikano, Keiji Kimura, Hironori Kasahara, "Performance Evaluation of Compiler Controlled Power Saving Scheme", Lecture Notes in Computer Science, Springer, Vol. 4759, pp.480-493, Jan. 2008.
- [6] Tomohiro Hirano, Hideo Yamamoto, Shuhei Iizuka, Kohei Muto, Takashi Goto, Tamami Wake, Hiroki Mikami, Moriyuki Takamura, Keiji Kimura, Hironori Kasahara, "Evaluation of Automatic Power Reduction with OSCAR Compiler on Intel Haswell and ARM Cortex-A9 Multicores", The 27th International Workshop on Languages and Compilers for Parallel Computing (LCPC), Sep. 2014.
- [7] Hironori Kasahara, Keiji Kimura, Toshiaki Kitamura, Hiroki Mikami, Kazutaka Morita, Kazuki Fujita, Kazuki Yamamoto, Tohma Kawasumi, "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, 2021, do: 10.1109/PEHC54839.2021.00007.
- [8] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep Residual Learning for Image Recognition. In Proceedings of the IEEE conference on computer vision and pattern recognition (pp. 770-778).
- [9] C. Lattner and V. Adve, "LLVM: A compilation framework for lifelong program analysis and transformation," in CGO, San Jose, CA, USA, Mar 2004, pp. 75-88.