エッジデバイス上でのメタデータ生成に向けた組込み用GPUを用いたニューラルネットワークの推論の高速化

寺倉　慶† 常　穹† 宮崎　純†

† 東京工業大学情報理工学院情報工学系　〒152–8550 東京都目黒区大岡山二丁目 12 番 1 号
E-mail: †terakura@lsc.c.titech.ac.jp, ††{q.chang,miyazaki}@c.titech.ac.jp

あらまし　本研究では, 組込みシステム用 GPU でのニューラルネットワークの推論を高速化する手法を提案する. 通常, 組込みシステム用 GPU でニューラルネットワークの推論を行う場合, TensorRT が使用されるが, TensorRT による量子化は一部のハードウェアでのみ対応している. そのため, 本研究では TensorRT を用いずに, CUDA を使用して重みとアクティベーションを量子化し, 汎用的な推論を実装する. 評価実験では, JetsonNano 上で CNN のモデルを TensorRT に変換したモデルを比較手法とし, 推論時間を比較した. 評価実験の結果, 精度を保ちつつ, 推論速度の高速化が可能であることが判明した.

キーワード　組込みシステム用 GPU, 量子化, 機械学習

1 はじめに

スマートフォン, スマート家電, 自律走行車など様々な機器に搭載されている組込みシステムでは, 画像認識, 言語翻訳, 意思決定など様々なタスクにニューラルネットワークが利用されることが多くなってきている. しかし, ニューラルネットワークを組込みシステムで利用する場合, 高速で効率的な処理や, リソースを効率的に使用することが重要であるという課題がある. これらの課題に対する解決策の 1 つとして, GPU（Graphics Processing Unit）を組込みシステムで使用することが挙げられる. GPU は, 並列処理能力が高いため, 従来の CPU（Central Processing Unit）と比較して, ニューラルネットワークの推論を大幅に高速化することが可能である. GPU は, 大量の並列処理を実現するために複数のコアを持ち, 各コアは独立して動作することができる. また, GPU は高速なメモリアクセス能力を持ち, 大量のデータを高速に読み書きすることができる. これらの特性により, ニューラルネットワークの推論に最適なハードウェアとされている. また, 現在では NVIDIA 社の JetsonNano などの組み込み向けのシステムに特化した GPU も開発されている. これらのシステムは, 小型かつ省エネルギーながらも高性能な GPU を搭載しており, 組込みシステムにおいても高速かつ効率的なニューラルネットワークの推論を実現することができる.

組込みシステム用の GPU でニューラルネットワークの推論を行う場合, PyTorch などのフレームワークで書かれたモデルを TensorRT で組み込みシステムに導入可能な形に変換し, 使用することが一般的である. TensorRT は, プルーニング, レイヤーフュージョン, 量子化など様々な最適化手法を実行でき, モデルサイズを大幅に縮小し, 推論速度を向上させることができる. しかし, TensorRT による最適化はすべてのデバイスに対応しているわけではなく, ターゲットとなるデバイスの性能によっては, 8 bit 量子化など一部の最適化技術が利用できない場合がある.

そのため, 本研究では, TensorRT を用いずに, CUDA を用いて量子化された推論のモデルを汎用化することで, TensorRT に依存することなく, TensorRT で変換したモデルよりもさらに高速な処理を実現する手法を提案する. 我々は, 固定小数点数を用いることで重みとアクティベーションをそれぞれ 8 bit に量子化することを実現した. また, バッチノーマライゼーション層では精度を保つために LOOKUP Table を使用した. LOOKUP Table を使用することで, 計算量を削減することができ, 推論の高速化にも繋がる. さらに, 畳み込み計算アルゴリズムとして, Winograd's minimal filtering algorithms を使用することで, 単純な畳み込みのアルゴリズムよりも高速な畳み込み演算を行うことができる.

本論文の手法は, CNN(Convolutional Neural Network) のモデルを使い, JetsonNano を用いて性能の評価を行う.

2 関連研究

2.1 CUDA

CUDA (Compute Unified Device Architecture) は NVIDIA によって開発された GPU 向けのプログラミングモデルであり, 高速な並列計算を可能にするために設計されている. CUDA は C/C++と互換性があり, 高度な並列処理を実現するための API を提供する. CUDA は, 画像処理, 機械学習, シミュレーションなどの様々なアプリケーションに使用されており, 特に深層学習においては非常に有用である.

CUDA プログラミングモデルでは, CUDA カーネルが GPU で実行される関数であり, これらの関数は複数の thread によって並列に処理される. thread のグループは block と呼ばれ, block は grid にグループ化される. block は 1 つのストリーミング マルチプロセッサ (SM) によって実行され, block に必要

なリソースに応じて, 1 つの SM で複数の block を同時に実行できる. また, 各ブロック内では, スレッドは 32 スレッドずつのワープに分割されている.

CUDA ではメモリを複数の種類に分けて使用する. GPU 全体で共有される大容量のメモリを global memory と言う. 全ての thread は, global memory からデータを読み取り, 書き込みができる. しかし, global memory へのアクセスは, 他のメモリへのアクセスに比べて時間がかかるという欠点がある. block 内の thread のみで共有することができるデータを保存するメモリを shared memory と言う. shared memory は使用できる容量が小さい代わりに, メモリアクセスが高速である.

さらに, GPU が効率よくメモリにアクセスするために, coalesced access と呼ばれる方法が存在する. coalesced access とは, 連続した thread が連続したメモリ領域に高速にアクセスすることであり, 分散されたメモリへのアクセスと比べ少ないトランザクションでメモリアクセスを実行できる.

2.2 TensorRT

TensorRT は, NVIDIA 社が提供する高速な推論エンジンである. TensorFlow や PyTorch などのフレームワークで訓練されたモデルを最適化することで, 高速な推論を実現することができる. TensorRT は不要なノードや層の削除, 量子化などを行い, モデルを最適化することで, モデルサイズを小さくすることができる. そのため, モデルをデプロイする際に必要なスペースや帯域幅を削減することができる. さらに, TensorRT は特定のハードウェアプラットフォーム (例: NVIDIA GPU) に最適化されており, ハードウェアに最適化されたモデルを生成することが可能である. TensorRT を使用するには, まず TensorFlow や PyTorch などで訓練されたモデルを TensorRT に対応した形式 (ONNX や UFF など) に変換する必要があり, その後, TensorRT API を使用してモデルを最適化し, 推論を行うことができる.

2.3 ニューラルネットワークの量子化

ニューラルネットワークの量子化は, 通常では高精度の値で演算を行うところを, 工夫して低精度の値で行うことを意味する. 量子化は, ニューラルネットワーク圧縮 [1] [2] のために最も広く使用されている技術の 1 つであり, 事前に訓練されたモデルを直接量子化する PTQ(Post-Training Quantization) [3] [4] と学習中に量子化誤差も含めて修正できるようにモデルを最適化する QAT(Quantization-Aware Training) [5] [6] の二種類に分類される. 本研究では PTQ を使用する. 量子化は精度と速度のトレードオフであり, QAT の場合, より精度を保ちつつ高速化することができるが, 複雑な技術や再トレーニングなどの時間が必要となる. また, GPU を使用する場合, 量子化を行うと, 計算速度が向上するだけでなく, グローバルメモリに対する転送量も減少するため, GPU にとっては大きな利点につながる.

2.4 固定小数点数

固定小数点数とは, 小数点以下の桁数が固定された数値のことである. 固定小数点数は, 浮動小数点数と比べて, 小数点以下の桁数が固定されているため, 表現できる精度が制限される. しかし, 固定小数点数は, 浮動小数点数よりも計算が高速であることが多いため, 計算量の多いプログラムでは, 固定小数点数を使うことで, 計算速度を向上させることができる. また, 固定小数点数を用いてニューラルネットワークを量子化する手法もたくさん提案されている [7] [8]. 以下, 固定小数点数の小数桁数を Q 表記を用いて表す. 例えば、小数桁数が 8 の場合は Q8 と記述する. a, b が固定小数点数の数でそれぞれ $Q\alpha$, $Q\beta$ とすると $\alpha > \beta$ の時, 固定小数点数同士の加算, 乗算は以下のようになる

- 加算

$$a(Q\alpha) + b(Q\beta) = a + b\ (Q\alpha) \tag{1}$$

- 乗算

$$a(Q\alpha) \times b(Q\beta) = a \cdot b\ (Q(\alpha + \beta)) \tag{2}$$

2.5 畳み込みニューラルネットワーク

畳み込みニューラルネットワーク (Convolutional Neural Network; CNN) は, 画像認識や自然言語処理などの様々なタスクに使われているアルゴリズムである. CNN では, 畳み込み層を複数重ねることで, 画像の特徴を抽出し, 分類や予測を行うことができる. 畳み込み層では, 畳み込みフィルタを用いて, 入力された画像をスキャンし, 画像の特徴を抽出する.

畳み込みフィルタは, 畳み込み層で入力された画像と重ね合わせることで, 画像の特徴を抽出するためのフィルタである. 畳み込みフィルタを用いて, 入力された画像をスキャンし, 畳み込み演算を行うことで, 画像の特徴を抽出し, 出力される特徴マップを生成する.

チャンネル数が C, フィルターのサイズが $R \times S$, 入力画像のサイズが $H \times W$ のとき, $Y_{i,k,x,y}$ を, i 番目のイメージに対する k 番目のフィルタの (x, y) 位置における出力を表し, $D_{i,c,x,y}$ を, i 番目のイメージの第 c チャンネルの (x, y) 位置における値を表し, $G_{k,c,u,v}$ を, k 番目のフィルタの第 c チャンネルの (u, v) 位置における値を表すとする. このとき畳み込み計算は以下のように表される.

$$Y_{i,k,x,y} = \sum_{c=1}^{C} \sum_{u=1}^{R} \sum_{v=1}^{S} G_{k,c,u,v} \cdot D_{i,c,x+u-1,y+v-1} \tag{3}$$

2.6 Winograd's minimal filtering algorithms

Winograd's minimal filtering algorithms は Shmuel Winograd によって最初に提案された画像処理において用いられる畳み込み演算の高速化アルゴリズムである [9]. 入力データを事前に変換し, 少ない乗算回数で畳み込み計算を行うことで, 高速化を実現している. Winograd's minimal filtering algorithms は, 畳み込みニューラルネットワークへの応用のためにも使われるようになり [10], 現在では NVIDIA が公開している Deep Learning 用のライブラリである cuDNN [11] で使用されているなど, 畳み込みニューラルネットワークに広く用いられている.

まず 1 次元の場合での Winograd Algorithm について説明する. サイズ n のフィルターを使用して, サイズ m の出力するための計算を $F(m, n)$ と表す. ここでは $F(2, 3)$ を例として考え

る. 単純アルゴリズムの場合, $F(2,3)$ を計算すると, $2 \times 3 = 6$ 回の乗算と 4 回の加算が必要となる. Winograd Algorithm では $F(2,3)$ を以下のように計算する.

$$F(2,3) = \begin{bmatrix} d_1 & d_2 & d_3 \\ d_4 & d_5 & d_6 \end{bmatrix} \begin{bmatrix} g_1 \\ g_2 \\ g_3 \end{bmatrix} = \begin{bmatrix} m_1 + m_2 + m_3 \\ m_2 - m_3 - m_4 \end{bmatrix} \quad (4)$$

このとき,

$$m_1 = (d_0 - d_2)g_0,\ m_2 = (d_1 + d_2)\frac{g_0 + g_1 + g_2}{2}$$
$$m_4 = (d_1 - d_3)g_2,\ m_3 = (d_2 - d_1)\frac{g_0 - g_1 + g_2}{2}$$

式 (4) は, 4 回の乗算と 8 回の加算が必要である. ただしフィルタの重みは入力に対して一定なので, $\frac{g_0+g_1+g_2}{2}$, $\frac{g_0-g_1+g_2}{2}$ は事前に計算可能である. つまり, 実際の加算の回数は 4 回となる. 元々, 6 回の乗算と 4 回の加算が必要だった計算が, 4 回の乗算と 4 回の加算で計算可能になっている.

Winograd Algorithm は, フィルタ g と入力タイル d の間の畳み込みを以下の式で表すことができる. $\odot$ をアダマール積 (要素ごとの積) とすると,

$$F(m,r) = A^T[(Gg) \odot (B^T d)] \quad (5)$$

行列 B, G, A は与えられた m と r の組み合わせに対して一定であり, $F(2,3)$ の場合は以下のようになる.

$$B^T = \begin{bmatrix} 1 & 0 & -1 & 0 \\ 0 & 1 & 1 & 0 \\ 0 & -1 & 1 & 0 \\ 0 & 1 & 0 & -1 \end{bmatrix},\ G = \begin{bmatrix} 1 & 0 & 0 \\ \frac{1}{2} & \frac{1}{2} & \frac{1}{2} \\ \frac{1}{2} & -\frac{1}{2} & \frac{1}{2} \\ 0 & 1 & 1 \end{bmatrix}$$

$$A^T = \begin{bmatrix} 1 & 1 & 1 & 0 \\ 0 & 1 & -1 & -1 \end{bmatrix}$$

$$g = \begin{bmatrix} g_0 & g_1 & g_2 \end{bmatrix}^T,\ d = \begin{bmatrix} d_0 & d_1 & d_2 & d_3 \end{bmatrix}^T$$

入力の値が 2 次元の場合, Winograd Algorithm は以下のように表すことができる.

$$F(m \times m, r \times r) = A^T[(GgG^T) \odot (B^T dB)]A \quad (6)$$

g は $r \times r$ のフィルターで, d は $(m + r - 1) \times (m + r - 1)$ の画像タイルである.

$F(2\times2, 3\times3)$ の場合, 単純なアルゴリズムだと, $2\times2\times3\times3 = 36$ 回の乗算が必要となる. Winograd Algorithm を使用した場合, G, B, A が定数であることを考慮すると, 必要となる乗算はアダマール積の部分だけなので $4 \times 4 = 16$ 回の乗算のみで計算可能である.

入力の値が 3 次元の場合, つまり, CNN での畳み込み計算と同じ場合, Winograd Algorithm は以下のように表すことができる. 入力の値のチャンネル数を C とすると,

$$F(m \times m, r \times r \times C) = \sum_{i=1}^{C} A^T[(Gg_iG^T) \odot (B^T d_i B)]A$$
$$= A^T[\sum_{i=1}^{C}(Gg_iG^T) \odot (B^T d_i B)]A \quad (7)$$

畳み込みニューラルネットワークで使用する場合は, 入力されるテンソルを $(m+n-1) \times (m+n-1) \times C$ のサイズのタイルに切り分けて, それぞれ計算し, 最終的に結合する (図 1).

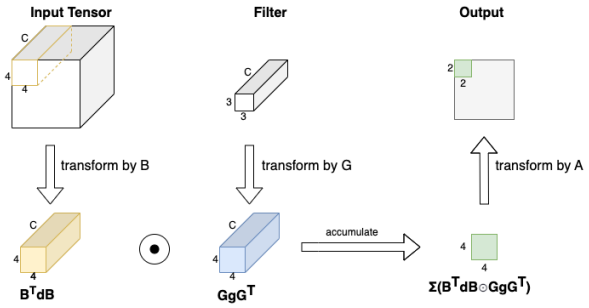

図 1　$F(2 \times 2, 3 \times 3 \times C)$ を Winograd Algorithm で計算する流れ

F(4, 3) の場合, B^T, G, A^T は以下のようになる.

$$B^T = \begin{bmatrix} 4 & 0 & -5 & 0 & 1 & 0 \\ 0 & -4 & -4 & 1 & 1 & 0 \\ 0 & 4 & -4 & -1 & 1 & 0 \\ 0 & -2 & -1 & 2 & 1 & 0 \\ 0 & 2 & -1 & -2 & 1 & 0 \\ 0 & 4 & 0 & -5 & 0 & 1 \end{bmatrix},\ G = \begin{bmatrix} \frac{1}{4} & 0 & 0 \\ -\frac{1}{6} & -\frac{1}{6} & -\frac{1}{6} \\ -\frac{1}{6} & \frac{1}{6} & -\frac{1}{6} \\ \frac{1}{24} & \frac{1}{12} & \frac{1}{6} \\ \frac{1}{24} & -\frac{1}{12} & \frac{1}{6} \\ 0 & 0 & 1 \end{bmatrix},$$

$$A^T = \begin{bmatrix} 1 & 1 & 1 & 1 & 1 & 0 \\ 0 & 1 & -1 & 2 & -2 & 0 \\ 0 & 1 & 1 & 4 & 4 & 0 \\ 0 & 1 & -1 & 8 & -8 & 1 \end{bmatrix}$$

F$(4 \times 4, 3 \times 3)$ を単純な畳み込みのアルゴリズムで計算すると, $4 \times 4 \times 3 \times 3 = 144$ 回の乗算が必要であるが, Winograd Algorithm では $6 \times 6 = 36$ 回の乗算で計算することができる. 大きなタイルの場合, 変換行列の要素の大きさも, タイルサイズの増加とともに大きくなる. 乗算の回数をより減らすことができるが, 行列の変換にかかるコストが大きくなってしまう.

3　提 案 手 法

本節では, CUDA で訓練済みのモデルを量子化し, 推論する方法について述べる. 本研究では, 固定小数点数を用いて, 重みとアクティベーションを Int8 に量子化し, 畳み込み計算は Int32 の精度で行っている. 最後にビットシフトを行うことで Int8 に変換し出力する (図 2).

本研究で使用するモデルは ResNet [12] をカスタマイズした

モデルであり, 本論文では, このモデルを ResNet-8(図 3) と命名する. conv3 と conv6 のフィルタのサイズは 1 × 1 であり, その他は 3 × 3 である.

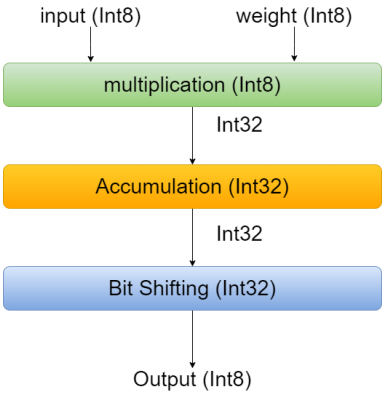

図 2　畳み込み計算の精度は Int32 で行いビットシフトを使って 8bit で出力

3.1　重みの量子化

本研究では訓練済みの重みを固定小数点数に変更し, それらを 8bit にして使用する. x を訓練済みの重み, n を固定小数点数の小数桁とすると, 重みを整数に変更する式は以下のようになる.

$$\lfloor x \cdot 2^n + 0.5 \rfloor \tag{8}$$

また Int8 にするために, −128 から 127 の範囲の外の値は, それぞれ −128, 127 に変換する. 式 (4) で得られた値を y とすると, 変換のための式は以下のようになる.

$$f(y) = \begin{cases} -128 & (y < -128) \\ y & (-128 \leqq y \leqq 127) \\ 127 & (y > 127) \end{cases} \tag{9}$$

3.2　畳み込み層

3.2.1　計 算 精 度

畳み込み層では, Int8 の入力と Int8 の重みを Int32 の精度で畳み込み計算を行う. 畳み込み計算をした後にビットシフト演算を行い, Int8 に戻す. ビットシフト演算による乗算, 徐算は切り捨てとなるので, 0.5 を足してからシフト演算を行うことで, 四捨五入する. 最後に式 (9) を適応して-128 から 127 の範囲に値を収める. 畳み込み層に入力される値の固定小数点数の小数桁数を m, 重みの小数桁数を n とすると, 式 (1), (2) より畳み込み計算後の小数桁数は $m + n$ になる. また, a を実数とすると畳み込み計算後の値を Int8 に戻すための式は以下のようになる.

$$(a \cdot 2^{m+n} + (1 << m-1)) >> m = \lfloor a \cdot 2^n + 0.5 \rfloor \tag{10}$$

3.2.2　Winograd Algorithm の使用

本研究では, $F(2 \times 2, 3 \times 3 \times C)$ の場合の, Winograd Algorithm を使用する. また, 畳み込みフィルタのサイズが $3 \times 3 \times C$ かつ, ストライドが 1 の時のみ, Winograd Algorithm を用いて計算する. つまり, 図 3 の Conv1, Conv2, Conv5, Conv6 で Winograd Algorithm を用いる. フィルタのサイズが $1 \times 1 \times C$ の場合, Winograd Algorithm は単純な畳み込みアルゴリズムと一致する. また, ストライドが 2 の場合は, Winograd Algorithm で出力された 4 つのピクセルのうち 1 つのピクセルしか出力結果には用いられず, 単純な畳み込みアルゴリズムと比べて無駄な計算が多くなってしまうため, Winograd Algorithm は用いない.

単純な畳み込みアルゴリズムで畳み込み計算を行う場合は, 1 block で, 1 枚の特徴マップを生成するよう設計にした. つまり, n 個のフィルターで畳み込みを行い n 枚の特徴マップを生成する場合は n block を使用して畳み込みを行う. また, 1 ピクセルの出力を 1 スレッドが計算し出力するように設計した.

3.2.3　Winograd Algorithm の実装方法

本研究では, 入力された $m \times m \times C$ のテンソルを $4 \times 4 \times C$ のサイズのタイルに分割し, 1 block で分割されたタイル 1 つを処理するように設計にした. この時タイルの数は $\frac{m}{2} \times \frac{m}{2}$ 個になる. タイリングの方法を図 4 に示す.

1 つの block 内で行われる計算は, まず初めに $B^T dB$ が計算され, 次に入力された GgG^T と $B^T dB$ のアダマール積をとり, 各チャンネルの和を計算する. これを入力された GgG^T の個数の数だけ行い, 最後に得られた行列を A を用いて変形する. つまり, Filter の数が n 個の場合, 1block で $2 \times 2 \times n$ のサイズの特徴マップを生成する (図 5). また, GgG^T は事前に計算しておき, その値を使う. 計算する際には固定小数点数で計算する必要があるため, G の値を $Q1$ として GgG^T を計算する. この時, GgG^T を 8 bit の精度で計算するとオーバーフローが起きてしまう可能性があるため, GgG^T は 16 bit の精度で計算し, 16 bit で保持する. つまり, Winograd Algorithm を用いて畳み込み計算を行うカーネルに渡される GgG^T は Int16 になる.

3.2.4　重みのシャッフル

本研究では値を深さ優先で保持し、重みを事前に図 6 のように入れ替えることで, 畳み込みの際に coalesced access を実現させている. ただし, output に加算する場合, 各フィルタと一回ずつ畳み込みをした後に, 書き込みの位置を一つずらす必要があるので, output に関しては完全に coalesced access ではない. カーネル内では output と input は shared memory を使用している. この操作を擬似コードで表すと Algorithm1 のようになる.

3.2.5　パディングの実装方法

一般的に畳み込み層では, 入力される特徴マップの端の情報が抜け落ちないように, 0 で特徴マップをパディングする. 本研究では, 0 で初期化されたパディング後のサイズの global memory を用意し, 出力を global memory に書き込む際に, 書き込む位置を調節することで実装した. また, 入力画像を入力とする最初のカーネルの時のみ, global memory から shared memory に

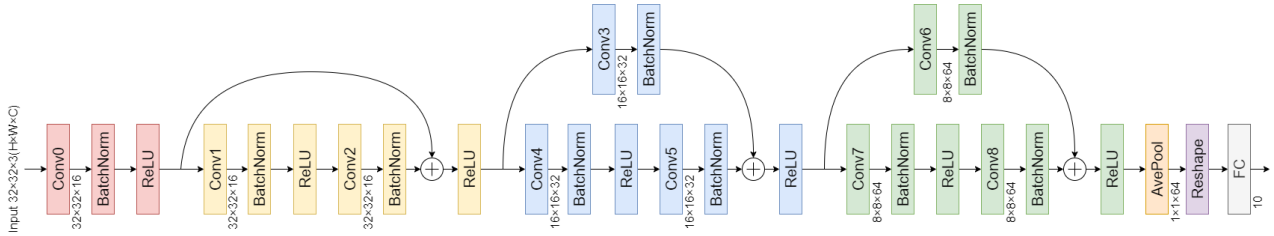

図 3　ResNet をカスタマイズした ResNet-8

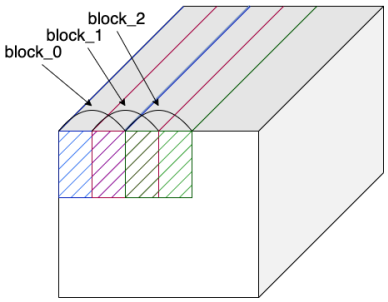

図 4　入力されたテンソルのタイリング方法

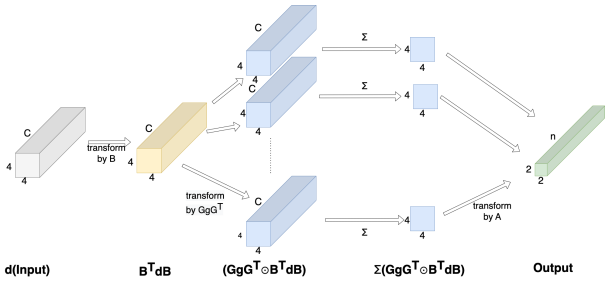

図 5　1 block 内で n 個のフィルタを用いて特徴マップを出力するまでの流れ

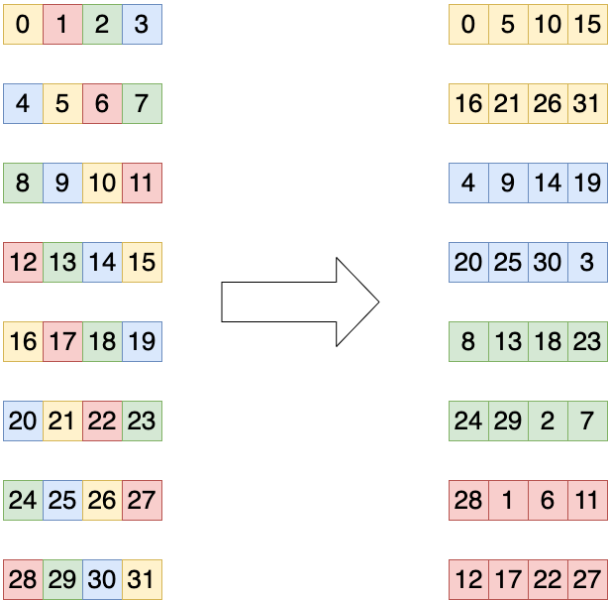

図 6　サイズ 1 × 1, チャンネル数 4, 個数 8 のフィルタのシャッフル

書き込む際にパディングを行った.

Algorithm 1 入力のチャンネル数が C, フィルタサイズ $1 \times 1 \times C$, 個数が n で $1 \times 1 \times n$ を出力する畳み込みの疑似コード

```
1: //smem は shared memory, gmem は global memory
2: //tid は thread id
3: __shared__ signed char input_smem[C];
4: __shared__ int output_smem[n];
5: input_smem ← input_gmem //入力を smem に書き込み
6: //畳み込み計算
7: for int i = tid; i < filter_size; i += block_size do
8:     in_ch = i%C //使用する入力のチャンネル
9:     out_ch = (i%n + i/n)%n //出力は n 回ごとに一つずらす
10:    output_smem[out_ch] += input_smem[in_ch] × filter[i]
11: end for
12: output_gmem ← output_smem //出力を gmem に書き込み
```

3.3　バッチノーマライゼーション層

推論時での, バッチノーマライゼーション層では x_i を入力されたデータ, μ_B をバッチ内の平均, σ_B^2 をバッチ内の分散, ϵ を微小な正の値, $\hat{x}_i$ を正規化されたデータ, γ と β を学習可能なパラメータ, y_i を BN 層の出力とすると, 以下のような計算が必要となる.

$$\hat{x}_i = \frac{x_i - \mu_B}{\sqrt{\sigma_B^2 + \epsilon}}, \quad y_i = \gamma \hat{x}_i + \beta$$

すべての値を固定小数点数で扱うと精度が低下してしまう. 本研究では, この精度低下の問題を解決するため, LOOKUP Table を用いて対応した. バッチノーマライゼーション層への入力の値は Int8 なので, 事前にバッチノーマライゼーション層への入力される可能性がある 256 個の固定小数点数の値を入力として, 単精度でバッチノーマライゼーションを計算し, その結果を Int8 に変換して LOOKUP Table に格納する. これにより, 精度を最大限に保つことができる. また, LOOKUP Table を用いることでバッチノーマライゼーション層の出力にかかる時間はメモリアクセスにかかる時間だけで済むので, 速度の向上も期待できる. さらに, バッチノーマライゼーション後に活性化関数を使用する場合は, LOOKUP Table に活性化関数を適応した後の値を格納しておくことで, バッチノーマライゼーションと活性化関数を 1 回のメモリアクセスのみで実現できる. LOOKUP Table の数は各層で出力されるチャンネル数と一致する.

3.4 カーネルの設定と役割

提案手法では global memory に対するアクセスをできるだけ減らすために, 畳み込み層, バッチノーマライゼーション層, アクティベーション層を 1 つのカーネル内で行った. さらに, 残差を計算する畳み込み層, バッチノーマライゼーション層は一つ前のカーネル内で行うようにした (図 7). 各層を 1 つのカーネルで実装することで, 中間データを global memory に書き出すことなく行えるため, メモリ転送にかかるコストを削減することができる.

4 評価実験

本章では, まず初めに, Winograd Algorithm を用いて畳み込み計算を行う手法と, 比較手法である単純な畳み込みアルゴリズムを用いて畳み込み計算を行う手法とを, 畳み込み計算自体の速度で評価する.

次に, 提案手法である ResNet-8 を量子化し, かつ, 畳み込み計算に Winograd Algorithm を使用した方法と, 比較手法である量子化は行わずに, 畳み込み計算には Winograd Algorithm を用いた方法, torch2trt [13] を用いて FP16 で TensorRT に変換する方法, torch2trt [13] を用いて FP32 で TensorRT に変換する方法とを推論時間で評価する.

4.1 DNN のモデル, 学習, 評価指標, パラメータ

データセットは 32 × 32 の RGB 画像 6 万枚, 10 クラスからなる Cifar10 [14] を使用する. 学習には SGD を用い, ミニバッチサイズは 32 である. 初期学習率は 0.075 であり, 学習率スケジューラとして cosine annealing を用いる. weight decay は 10^{-4} であり, momentum は 0.9 である. エポック数は 500 である. 各手法で, 推論精度と推論時間を評価した. 推論精度は, CIFAR10 データセットのテストセットで測定した. 推論時間は, 入力が GPU に転送された状態から出力が CPU に転送されるまでの時間である.

4.2 実験環境

Quad-core ARM A57 プロセッサ, 921MHz で動作する 128 コア NVIDIA Maxwell GPU, 4GB メモリを搭載した Jetson Nano を使用した. JetPack 4.6.1 を使用した.

4.3 実験内容

CUDA を使って量子化し, Winograd Algorithm を用いて畳み込み計算を行うモデル, Winograd Algorithm を用いるが, 量子化は行わずにすべて単精度で計算されるモデル, TensorRT を用いて最適化されたモデルの性能を比較する. それぞれ, 事前に FP32 の精度で学習した Pytorch モデルを最適化する. TensorRT では推論タスクの最適化手法として, torch2trt を用いて, Pytorch モデルを精度が FP16 の TensorRT へ変換した. 推論時間の評価にはランダムで生成された入力に対して 10 回推論時間を求め, その平均を求めた. ウォームアップとして事前に 500 回推論した状態から実験を行なっている. 固定小数点数の精度は入力される画像を $Q7$, 各層に入力, 出力される特徴量を $Q3$, 重みを $Q5$ として計算した.

Winograd Algorithm の評価に関しては, 単純な畳み込みアルゴリズムで, Winograd Algorithm の実装と同じく, 入力をタイリングして, 1 block で $2 \times 2 \times C$ の特徴マップを出力するような実装と比較した. 評価に使用する入力, フィルター, 出力のサイズは以下の三組である.

- Input: $(32 \times 32 \times 16)$, Filter: $(32 \times 32 \times 16 \times 16)$, Output: $(32 \times 32 \times 16)$
- Input: $(16 \times 16 \times 32)$, Filter: $(16 \times 16 \times 32 \times 32)$, Output: $(16 \times 16 \times 32)$
- Input: $(8 \times 8 \times 16)$, Filter: $(8 \times 8 \times 64 \times 64)$, Output: $(8 \times 8 \times 16)$

ウォームアップとして事前に 100 回畳み込みを行った状態から実験を行なっている.

5 実験結果

5.1 Winograd Algorithm の評価の実験結果

入力サイズが $(a \times a \times m)$, 出力サイズが $(b \times b \times n)$ の畳み込みを in_a_m_out_b_n と表す. 提案手法と比較手法の実行速度の違いを比較を表 1 に示す.

in_32_16_out_32_16 では約 1.31 倍, in_16_32_out_16_32 では約 1.52 倍, in_8_64_out_8_64 では約 1.50 倍, 速度が向上した.

表 1 Winograd Algorithm と単純な畳み込みアルゴリズムとの比較 (単位 [ms])

手法	in32_16_out32_16	in16_32_out16_32	in8_64_out8_64
Winograd	0.200	0.203	0.160
単純畳み込み	0.261	0.309	0.240

5.2 量子化の評価の実験結果

提案手法と比較手法の推論時間と精度の比較を表 2 に示す. 実験結果より, 提案手法は比較手法である TensorRT を用いたモデルと比較して推論速度が約 1.06 倍に向上し, 精度は約 2 % の低下にとどめることができた. 量子化されていない提案手法のモデルと量子化された提案手法を比べると, 推論速度が約 2.19 倍となった. また, 量子化されていない提案手法と量子化されていない比較手法を比べると, 推論速度は約 0.59 倍となった.

表 2 入力に同じ値を使った場合の提案手法と比較手法 (TensorRT) との比較

手法	データ型	Acc[%]	推論時間
提案手法 (量子化あり)	INT8	87.21	1.083
提案手法 (量子化なし)	FP32	89.61	2.369
比較手法 (TensorRT)	FP16	89.58	1.147
比較手法 (TensorRT)	FP32	89.58	1.402

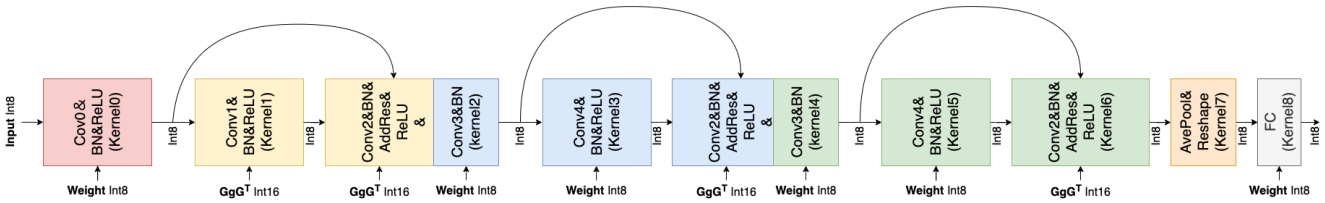

図 7　ResNet-8 を GPU を使って推論する際の流れと各カーネルの役割

5.3 考　　察

Winograd Algorithm については, 乗算の回数の減少が速度の向上につながったといえる. 入力のサイズによって速度の変化率が異なるのは, GPU の資源の使用率などが影響してきていると考えられる.

量子化については, 実験結果より, 提案手法では, 重みとアクティベーションを 8 bit に量子化することによって, 推論速度が高速化されたと考えれられる. TensorRT を使わずに, INT8 まで計算精度を落とすことにより, 計算速度やメモリの転送速度が上がり, 推論の高速化につながったといえる. 提案手法のうち, 量子化されたモデルとされていないモデルの差は, 計算速度や, メモリの転送速度だけではなく, バッチノーマライゼーション層での LOOKUP Table の有無も影響していると考えられる. また, 量子化されていないモデルの提案手法と比較手法の実験結果から, 量子化以外の部分で提案手法はさらに最適化の余地があることがわかる.

推論の精度に関して, 今回の実験ではすべての層のフィルターで固定小数点数の小数桁を同一にしているため, 必要以上に精度を落としている可能性がある. 各層のフィルタの値の幅に合わせて, 扱う小数桁を変更することで, 実験で得られた以上の精度を保つことができると考えられる.

実験結果から, 提案手法は TensorRT に比べてより高速化が可能であることが示され, 速度面では TensorRT は不要にも見える. ただし, この評価は ResNet のみに限られており, 一般化することはできない. ほかの CNN のモデルに対しても同様の結果が得られるかどうかはさらなる実験が必要である.

6 おわりに

本研究では, 組込みシステム用 GPU 上での CNN の推論速度の高速化を目的として, GPU 上で Winograd Algorithm を実装する方法と TensorRT を用いずに, CUDA を用いて重みとアクティベーションを量子化する汎用的な方法を提案した. 評価実験では JetsonNano を用いて, Winograd Algorithm と単純な畳み込みアルゴリズムの畳み込み計算にかかる速度を比較した. 評価実験の結果から, Winograd Algorithm を使った畳み込みでは, 単純な畳み込みアルゴリズムに比べて, 計算の速度が約 1.31 倍から約 1.51 倍まで向上した. また, ResNet の推論速度の比較も行った. 事前に FP32 の精度で学習した Pytorch モデルを torch2trt を用いて TensorRT に変換し最適化したモデルと, Winograd Algorithm を使って畳み込みを行い, さらに量子化したモデルを比較した. 推論速度に関しては, TensorRT を使用して FP16 で推論したモデルに比べ, 精度の低下を約 2 % にとどめ, 推論速度を約 1.06 倍高速化することに成功した. これらの結果より, ResNet のモデルでの推論を TensorRT による INT8 への量子化が対応していない組込み時システム用 GPU を用いる場合は, CUDA を用いて重みとアクティベーションを量子化することで推論の高速化を実現できることが分かった.

今後の課題として次のような事項が挙げられる. ResNet 以外の CNN のモデルに提案手法を適応する. 本研究では ResNet での実験しか行っておらず, ほかの CNN のモデルでも同様の結果が得られるか確認する必要がある. また, GPU の資源をより効率的に使用できる実装に変更することも重要である. タイリング方法を工夫し, GPU の資源をフルに活用できる仕様にすることでさらなる速度の向上が見込める. その他には, 巨大なモデルへの対応も必要である. 巨大なモデルの場合だと, 本研究で使用した LOOKUP Table の数や, Winograd Algorithm で使用する重みの数が多くなり, GPU 上のメモリ不足になる可能性が考えられる. そのため, LOOKUP Table を用いない手法や工夫を考える必要がある.

謝　　辞

本研究は, JST CREST JPMJCR22M2 の支援を受けたものである.

文　　献

[1] Matthieu Courbariaux, Yoshua Bengio, and Jean-Pierre David. Advances in neural information processing systems. *Proceedings of the 28th International Conference on Neural Information Processing Systems*, Vol. 2, No. 10, p. 3123–3131, 2015.

[2] Song Han, Huizi Mao, and William J Dally. Deep compression: Compressing deep neural networks with pruning amd trained quantization and huffman coding. *arXiv preprint arXiv:1510.00149*, 2015.

[3] Markus Nagel, Mart van Baalen, Tijmen Blankevoort, and Max Welling. Data-free quantization through weight equalization and bias correction. *Proceedings of the IEEE/CVF International Conference on Computer Vision (ICCV)*, p. 1325–1334, October 2019.

[4] Jun Fang, Ali Shafiee, Hamzah Abdel-Aziz, David Thorsley, Georgios Georgiadis, and Joseph H. Hassoun. Post-training piecewise linear quantization for deep neural networks. *Computer Vision – ECCV 2020*, pp. 69–86, October 2020.

[5] Benoit Jacob, Skirmantas Kligys, Bo Chen, Menglong Zhu, Matthew Tang, Andrew Howard, Hartwig Adam, and Dmitry Kalenichenko. Quantization and training of neural networks for efficient integer-arithmetic-only inference. pp.

2704–2713, June 2018.

[6] Philipp Gysel amd Jon Pimentel amd Mohammad Motamedi amd Soheil Ghiasi. Ristretto: A framework for empirical study of resource-efficient inference in convolutional neural networks. *IEEE Transactions on Neural Networks and Learning Systems*, Vol. 29(11), pp. 5784 – 5789, November 2018.

[7] Darryl Lin amd Sachin Talathi and Sreekanth Annapureddy. Fixed point quantization of deep convolutional networks. *International conference on machine learning*, Vol. PMLR, p. 2849–2858, 2016.

[8] Rishabh Goyal, Joaquin Vanschoren, Victor van Acht, and Stephan Nijssen. Fixed-point quantization of convolutional neural networks for quantized inference on embedded platforms. *CoRR*, Vol. abs/2102.02147, , 2021.

[9] S Winograd. *Arithmetic Complexity of Computations*, Vol. 33. 1980.

[10] Andrew Lavin and Scott Gray. Fast algorithms for convolutional neural networks. pp. 4013–4021, June 2016.

[11] Sharan Chetlur, Cliff Woolley, Philippe Vandermersch, Jonathan Cohen, John Tran, Bryan Catanzaro, and Evan Shelhamer. cudnn: Efficient primitives for deep learning. *CoRR*, 2014.

[12] 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 (CVPR)*, pp. 770–778, June 2016.

[13] NVIDIA-AI-IOT. torch2trt: An easy to use pytorch to tensorrt converter. `https://github.com/NVIDIA-AI-IOT/torch2trt`.

[14] Alex Krizhevsky, Vinod Nair, and Geoffrey Hinton. Cifar-10. `http://www.cs.toronto.edu/~kriz/cifar.html`.