Bulletin of Networking, Computing, Systems, and Software – www.bncss.org, ISSN 2186-5140
Volume 8, Number 1, pages 45–49, January 2019

Fast Algorithm of Unified Layer Performing Convolution and Average Pooling on the GPU

Hiroki Tokura, Takahiro Nishimura, Yasuaki Ito, and Koji Nakano
Department of Information Engineering, Hiroshima University
Kagamiyama 1-4-1, Higashi-Hiroshima, 739-8527 JAPAN

Akihiko Kasagi, and Tsuguchika Tabaru
FUJITSU LABORATORIES LTD.
4-1-1 Kamikodanaka, Nakahara-ku, Kawasaki, Kanagawa 211-8588, JAPAN

***Abstract*—Recently, Convolutional Neural Networks (CNN) have made a major contribution in the field of recognition. CNN has multiple convolution layers and convolution operations are needed large number of floating-point operations. So, convolution operations are bottleneck of CNN. The main contribution of this paper is to present new methods for convolution and average pooling computation on the GPU. First, we present fused filter method. An average pooling can be considered as a kernel. A convolution kernel and an average kernel can be fused. In fused filter method, convolution using fused filter reduces floating-point operations for convolution and pooling computation. Also, we present direct sum method. Convolution and average pooling computation are commutative. In direct sum method, switching convolution and average pooling computation reduces floating point operations for convolution and pooling computation. Experimental results using NVIDIA V100 show direct sum method attains a speed-up factor of up to 1.8 (single precision) and 4.4 (half precision) over cuDNN naive implementation.**

***Index Terms*—Deep Learning, Neural Network, Convolution, Average Pooling, GPU**

I. はじめに

GPU は画像処理や 3D グラフィックスなどを高速に処理することができ，NVIDIA 社は CUDA [8] と呼ばれる GPU における並列プログラミング環境を提供している．また，高度にチューニングされた線形代数ライブラリの cuBLAS [7] や Deep Learning 用ライブラリの cuDNN [9] などのライブラリも提供されている．畳み込みニューラルネットワーク (CNN) は画像分類に用いられる一般的なネットワークの一つである [4]．近年，CNN を用いたディープニューラルネットワークの精度が向上している．

CNN は畳み込み層とプーリング層のペアを複数回用いて構成される場合が多い．入力 X に対して畳み込みを行った結果 Z に対してプーリングを行うことで入力の特徴パターン Z を抽出する．特徴抽出を行うには複数の畳み込みカーネルを必要とするため，畳み込みの計算コストが増大する．プーリング層では畳み込み層で抽出された特徴の位置感度を減少させてプーリング層の出力を普遍にするとともに，特徴パターンの空間使用量を減少させる．プーリング層では最大値プーリングまたは平均値プーリングのどちらかが用いられるが，本論文では線形の結果が得られる平均値プーリングに焦点を当てる．以下はサイズ $n \times n$ の入力 X, サイズ $k \times k$ の畳み込みカーネル W, 入力チャネル数 c_i, 出力チャネル数 c_o, 平均値プーリングサイズ $p \times p$ の際の畳み込み層，平均値プーリング層それぞれにおけるナイーブな計算方法である．

[畳み込みカーネル W を用いた入力 X に対する畳み込み結果 Y]

```
for x ← 0 to c_o − 1 do
 for i ← 0 to n − k + 1 do
  for j ← 0 to n − k + 1 do
   Y[x][i][j] ← 0
   for c ← 0 to c_i − 1 do
    for m ← 0 to k − 1 do
     for n ← 0 to k − 1 do
      Y[x][i][j] ← I[c][i + m][j + n]W[x][c][m][n]
                   +Y[x][i][j]
```

[プーリングサイズ $p \times p$ における Y に対する平均値プーリング結果 Z]

```
for x ← 0 to c_o − 1 do
 for i ← 0 to (n−k+1)/p do
  for j ← 0 to (n−k+1)/p do
   Z[x][i][j] ← 0
   for k ← 0 to p − 1 do
    for l ← 0 to p − 1 do
     Z[x][i][j] ← Z[x][i][j] + Y[x][p × i + k][p × j + l]
   Z[x][i][j] ← p^{−2} Z[x][i][j]
```

II. 関連研究

CNN の計算量を削減するための手法がいくつか提案されている．計算量を削減する手法の 1 つは，FFT を用いて畳み込みを行う手法である [6]．本アルゴリズムではアルゴリズムの計算量が畳み込みカーネルの大きさに依存しないため，畳み込みカーネルサイズの大きな場合に本アルゴリズムは有用である．FFT アルゴリズムの他に Winograd アルゴリズムも提案されている [5]．本アルゴリズムは畳み込みカーネルサイズが十分に小さい場合ナイーブな畳み込み計算よりも計算量を削減することができ，また FFT よりも高速に計算可能である．

論文 [3] では，CPU での実装のみだが，我々の研究と同様に畳み込み層と平均値プーリング層の 2 つの処理全体に焦点を当て，そこに Summed Area Tabel [2] 手法を用いることで高速な計算を可能にしている．

III. 提案手法

本章では畳み込み層と平均値プーリング層の連続する 2 つの層をまとめることで計算量を削減する 2 つの手法について説明する．手法としては，平均値プーリング計算も 1 つの畳み込みカーネルとみなし，畳み込みカーネルとプー

リングカーネルを畳み込んで 1 つのカーネルにしてしまう Fused Filter 手法と，平均値プーリングと畳み込みの順番を入れ替えて計算を行う Direct Sum 手法を提案する．

A. *Fused Filter* 手法

サイズ $p \times p$ の平均値プーリングを行うことは，全ての要素が $\frac{1}{p^2}$ であるサイズ $p \times p$ のプーリングカーネルを用いて畳み込みを行うことと同義である．

$p \times p$ 平均値プーリングは $p \times p$ サイズの畳み込みカーネル W_p で置き換えることが可能である. W_p による畳み込みは，入力に対して水平，垂直方向に対して p 要素ごと行う. p 要素ごとの畳み込み操作を以後，ストライド p と呼ぶ. また，畳み込みカーネル W_c と W_p は交換法則，結合法則が成り立つ. 入力 X に対してカーネル W_c, W_p を順番に畳み込んだ結果と，カーネル W_c, W_p を畳み込んだ W_{cp} を新たなカーネルとして入力 X に対してストライド p で畳み込んだ結果は同様のものになる．

以上の考えを用いると，畳み込みカーネル W_c とプーリングカーネル W_p を畳み込んでその結果 W_{cp} を用いてストライド p で入力 X を畳み込むことで，ナイーブに畳み込みとプーリングを行う場合と同様の結果が得られる．本論文では畳み込みカーネルとプーリングカーネルを畳み込んだものを Fused Filter と呼ぶ．また Fused FilterW_{cp} を用いてストライド p で畳み込みを実行することで畳み込み層とプーリング層をまとめることが可能である．Fused Filter を用いて入力 X から畳み込みと平均値プーリングの結果 Z を直接計算するアルゴリズムを以下に示す．

[Fused Filter を用いた畳み込みと平均値プーリング]
for $x \leftarrow 0$ to $c_o - 1$ do
 for $i \leftarrow 0$ to $\frac{n-k+1}{p}$ do
  for $j \leftarrow 0$ to $\frac{n-k+1}{p}$ do
   $Z[x][i][j] \leftarrow 0$
   for $c \leftarrow 0$ to $c_i - 1$ do
    for $k \leftarrow 0$ to $k+p-1$ do
     for $l \leftarrow 0$ to $k+p-1$ do
      $Z[x][i][j] \leftarrow X[c][p \times i + k][p \times j + l] W_{cp}[x][c][k][l] + Z[x][i][j]$

B. *Direct Sum* 手法

CNN において，畳み込み層の後にプーリング層を置き，それを繰り返すことで学習を行っていくのが一般的である．ここで畳み込み層とプーリング層の順番を入れ替えることで，中間出力 Y を計算することなく直接入力 X から出力 Z を計算することが可能である．入力 X のサイズを $n \times n$，畳み込みカーネル W のサイズを $k \times k$，平均値プーリングサイズを $p \times p$ とすると，畳み込み後の中間出力 Y のサイズは $(n-k+1) \times (n-k+1)$，最終出力 Z のサイズは $\frac{n-k+1}{p} \times \frac{n-k+1}{p}$ となる．ここで，$c_i = 1, c_o = 1, k = 3, p = 2$ とすると，最終出力の 1 要素 $Z[0][0][0]$ は以下のように計算される．

$$Z[0][0][0] = \sum_{0 \le k < 2, 0 \le l < 2} Y[0][k][l] \times \frac{1}{2^2}$$

また中間出力 Y の 1 要素 $Y[0][k][l]$ は以下のように計算される．

$$Y[0][k][l] = \sum_{0 \le i < 2, 0 \le j < 2} W[0][i][j][0] \times X[0][i+k][j+l]$$

この式から畳み込みカーネル W は中間出力 Y のインデックスとは依存関係がないことがわかる．よって最終出力の 1 要素 $Z[0][0][0]$ は以下のように計算可能である．

$$\begin{aligned} Z[0][0][0] &= \sum_{\substack{0 \le k < 2 \\ 0 \le l < 2}} \sum_{\substack{0 \le i < 2 \\ 0 \le j < 2}} W[0][i][j][0] \times X[i+k][j+l][0] \times \frac{1}{2^2} \\ &= \sum_{\substack{0 \le i < 2 \\ 0 \le j < 2}} W[0][i][j][0] \left(\sum_{\substack{0 \le k < 2 \\ 0 \le l < 2}} X[i+k][j+l][0] \right) \times \frac{1}{2^2} \end{aligned}$$

上記の式を見ると分かる通り，入力 X 内の $X[i][j]$ を左上の要素と見た 2×2 の矩形の総和と畳み込みカーネル W の対応する要素を掛け合わせ，それらの結果を総和することで，中間出力 Y を計算することなく最終出力 Z を求めることができる．そこで提案手法では，入力 X に対して横方向と縦方向に順番に一次元の加算を行って必要な矩形総和を求めた後に W で畳み込みを行うことで最終出力 Z を得る．この手法を Direct Sum 手法と呼ぶ．Direct Sum 手法のアルゴリズムを以下に示す．

[Direct Sum を用いた畳み込みと平均値プーリング]
for $x \leftarrow 0$ to $c_o - 1$ do
 for $i \leftarrow 0$ to $n-2$ do
  for $j \leftarrow 0$ to $n-2$ do
   $y \leftarrow 0$
   for $a \leftarrow 0$ to $p-1$ do
    for $b \leftarrow 0$ to $p-1$ do
     $y \leftarrow y + X[i+a][j+b]$
   $X[x][i][j] \leftarrow y p^{-2}$
for $x \leftarrow 0$ to $c_o - 1$ do
 for $i \leftarrow 0$ to $\frac{n-k+1}{p}$ do
  for $j \leftarrow 0$ to $\frac{n-k+1}{p}$ do
   $Z[x][i][j] \leftarrow 0$
   for $c \leftarrow 0$ to $c_i - 1$ do
    for $k \leftarrow 0$ to $k-1$ do
     for $l \leftarrow 0$ to $k-1$ do
      $Z[x][i][j] \leftarrow X[c][p \times i + k][p \times j + l] W[x][c][k][l]$

Fused Filter を用いたアルゴリズムと比較してみると，Direct Sum 手法は矩形総和を求める処理が増えているものの畳み込み部分のループ回数が減少していることがわかる．以下で Fused Filter 手法と Direct Sum 手法それぞれの浮動小数点演算数を比較する．

ナイーブな畳み込みでは、1 要素の畳み込みに $2k-1$ 演算が必要となる. すべての要素に対しては $(2k^2-1)n^2 c_i$ 演算が必要となる. さらに，チャンネル方向の総和を求める必要があるため，結果として $(2k^2-1)n^2 c_i + (c_i - 1)n^2$ 演算が必要である. また，出力チャンネルは c_o であるので，$((2k^2-1)n^2 c_i + (c_i-1)n^2)c_o$ 演算ですべての畳み込みを計算できる. 平均値プーリングでは，各要素に対して p 演算が必要となる. p 要素ごとでプーリングを行うので，出力チャンネルも考慮したすべての要素に対しては，$p^2(\frac{n}{p})^2 c_o$ 演算が必要である. そのため，ナイーブな畳み込みと平均値プーリングでは，$2k^2 c_i c_o n^2 + p^2(\frac{n}{p})^2 c_o$ 演算が必要である. また，Winograd アルゴリズムでは，必要な演算数を削減することができる [5]. このアルゴリズムにおいて，入力の 2×2 要素に対する 3×3 カーネルによる畳み込み演算数はカーネルの変換に 36, 畳みこまれるデータの変換に 32, 畳み込みの演算に 36 演算が必要となる. しかしながら，カーネルの変換は一度変換を行えば再利用することができる. そのため，すべ

ての要素の畳み込みには, $36c_ic_o + 68\left(\frac{n}{2}\right)^2(c_i-1)c_o$ 演算が必要となる. Winograd アルゴリズムによる畳み込みと平均値プーリングでは，$36c_ic_o + 68\left(\frac{n}{2}\right)^2(c_i-1)c_o + p^2(\frac{n}{p})^2c_o$ 演算が必要である.

Fused Filter 手法では，サイズ $(k+p-1)\times(k+p-1)$ の畳み込みカーネルによりストライド p により畳み込みが行われる. そのため，$(2(k+p-1)^2c_i-1)(\frac{n}{p})^2c_o$ 演算が必要である.

Direct Sum 手法では，はじめに各チャンネルの $p\times p$ 要素の合計を求める必要がある. そして，サイズ $k\times k$ のカーネルによるストライド p による畳み込みが行われる. そのため，Direct Sum 手法では $2pn^2c_i + (2k^2c_i-1)c_o(\frac{n}{p})^2$ 演算が必要である. 各手法に必要な浮動小数点数演算数を表 I にまとめる.

Direct Sum 手法が浮動小数点数演算数が最も少なく，ついで Fused Filter 手法，ナイーブ手法 (Winograd), ナイーブ手法となる. $n=32, k=3, c_i=512, c_o=512, p=2$ の場合では，Direct Sum 手法による浮動小数点数演算数はナイーブ実装に比べ約 4 倍少なく，Fused Filter 手法に対しては約 1.8 倍少ない.

TABLE I
各手法に必要な浮動小数点数演算数

手法	浮動小数点演算数
naive 手法	$2k^2c_ic_on^2 + p^2(\frac{n}{p})^2c_o$
naive 手法 (Winograd)	$36c_ic_o + 68\left(\frac{n}{2}\right)^2(c_i-1)c_o + p^2(\frac{n}{p})^2c_o$
Fused Filter 手法	$(2(k+p-1)^2c_i-1)(\frac{n}{p})^2c_o$
Direct Sum 手法	$2pn^2c_i + (2k^2c_i-1)c_o(\frac{n}{p})^2$

IV. GPU 実装

本章では，初めに CUDA について説明を行った後に，提案手法である Fused Filter 手法と Direct Sum 手法の GPU 実装について説明する．実装には NVIDIA 社が提供する Deep Learning 用ライブラリの cuDNN [9] 及び線形代数ライブラリの cuBLAS [7] を用いる．

NVIDIA 社は NVIDIA 社製 GPU 向けに CUDA と呼ばれる並列計算のためのソフトウェアを提供している. CUDA では，グローバルメモリとシェアードメモリで構成されており，グローバルメモリは大容量だが低速であり，シェアードメモリは小容量だが高速である [12]. CUDA 並列プログラミングモデルでは，グリッド，スレッドブロック，スレッドと呼ばれるスレッドの階層構造を持っている. 1 グリッドはスレッド数の等しい複数のスレッドブロックを持ち，スレッドブロックは Streaming Multiprocessor(SM) に割り当てられ，スレッドブロック内のすべてのスレッドは同じ SM で並列に実行される. すべてのスレッドはグローバルメモリにアクセス可能だが，ブロック内のスレッドは割り当てられた SM のシェアードメモリにのみアクセス可能である. CUDA では，ユーザーが C 言語を拡張した CUDA C を用いてカーネルと呼ばれる関数を定義可能である. カーネルを起動することで，スレッドブロックは SM に割り当てられ，スレッドブロック内のスレッドは同じ SM 内のコアに割り当てられる. グローバルメモリやシェアードメモリを適切に使用することで高いパフォーマンスを発揮できる [11]. しかしながら，任意のアプリケーションにおいて常に高いパフォーマンスを発揮することは困難であるため，NVIDIA 社が提供している高度にチューニングされたライブラリを使用する.

cuBLAS [7] は，NVIDIA 社製 GPU にきわめてチューニングされた線形代数ライブラリである. cuBLAS では，行列ベクトル積，行列積や LU 分解など様々な関数が提供されている. また，行列積の関数では，倍精度，単精度，半精度浮動小数点数の計算がサポートされており，Volta 世代の GPU から搭載された半精度浮動小数点数の行列積を高速に行う演算器であるテンソルコアを用いた計算もサポートされている. cuDNN [9] は，cuBLAS と同じように，チューニングされたディープラーニング向けライブラリである. cuDNN では，全結合層，畳み込み層，プーリング層といった関数を提供している. 畳み込み計算では行列積を用いた畳み込み計算, winograd アルゴリズムを用いた畳み込み計算, FFT を用いた畳み込み計算が実装されており, cuDNN は入力サイズやカーネルサイズから自動的に適切なアルゴリズムにより計算が行われる. また, テンソルコアを用いた計算も多くの関数がサポートされている.

A. *cuDNN* 実装

cuDNN を用いて Naive 手法を実装する場合，cuDNN の畳み込み関数とプーリング関数を実行することで計算が完了する. cuDNN を用いて Fused Filter 手法を実装する場合，cuDNN に直接 Fused Filter を引数として指定して実行することで最終出力を得る. そのため，cuDNN の畳み込み関数の実行のみで計算が完了する.

Direct Sum 手法を実装する場合，入力 X の矩形総和を求めた結果を入力として cuDNN に渡し，ストライド p で畳み込みを実行することで最終出力を得る. 矩形総和を求める処理を cuDNN と作成した GPU カーネルでの実行を比べたところ，GPU カーネルが cuDNN より高速であった. そのため，矩形総和を求める処理は GPU カーネルで行う. そのため，CUDA カーネルと cuDNN の畳み込み関数の実行で計算が完了する. cuDNN において，畳み込みを行う場合は *cudnnConvolutionForward*，プーリングを行う場合は *cudnnPoolingForward* を用いた. また，半精度浮動小数点数でテンソルコアを用いて計算を行う場合は *cudnnSetConvolutionMathType* により *CUDNN_TENSOR_OP_MATH* を指定した.

B. *cuBLAS* 実装

一般的に畳み込み計算を行列積に変換することで高速に計算することが可能である [1]．そこで行列積を高速に計算可能な cuBLAS を用いる．

Fused Filter 手法を実装する場合，まず入力 X を行列積に適した形式に変換する．畳み込みカーネルサイズ $k\times k$, プーリングサイズ $p\times p$ において入力を行列積に適した形式に変換する際，使用メモリ量が $\left(\frac{k}{p}\right)^2$ 倍に増加する．変換に用いる GPU 用の関数を定義し，変換関数を実行したのちに変換後の行列と Fused Filter を入力として cuBLAS を実行することで最終出力を得る. そのため，CUDA カーネルと cuBLAS の行列積計算関数の実行で計算が完了する. Direct Sum 手法の実装には入力 X における矩形総和の計算及び行列積の形式への変換が必要だが，この 2 つの処理を実行する GPU 用の関数を実装した．

そのため，CUDA カーネルと cuBLAS の行列積計算関数の実行で計算が完了する. 単精度浮動小数点数で計算を行う場

合は *cublasSgemm*，半精度浮動小数点数で計算を行う場合は *cublasHgemm* を用いて行列積を計算した．また，半精度浮動小数点数でテンソルコアを用いて計算を行う場合は *cublasSetMathMode* により *CUBLAS_TENSOR_OP_MATH* を指定し，*cublasGemmEx* を用いて行列積を計算した．また，引数の *algo* には *CUBLAS_GEMM_DEFAULT_TENSOR_OP* を指定した．

V. 実験結果

本章では実験結果について述べる．cuDNN を用いた畳み込みと平均値プーリングをナイーブに行うナイーブ実装，cuDNN を用いた Fused Filter 手法と Direct Sum 手法，cuBLAS を用いた Fused Filter 手法と Direct Sum 手法のそれぞれを NVIDIA Tesla V100 GPU に実装した．また，CUDA9.2，cuDNN7.3 を用いた．入力サイズ，畳み込みカーネルサイズ，プーリングサイズはそれぞれ $32\times32, 3\times3, 2\times2$ とした．入力チャネル数及び出力チャネル数は 32~512 とした．cuDNN の畳み込みでは，基本的に Fused Filter 手法と Direct Sum 手法では行列積が用いられていた．1000 回実行を繰り返し，その平均値を実行時間とした．また，計測には CUDA イベントタイマーを使用した．このタイマーの時間分解能はおよそ $0.5\mu s$ である [10].

表 II は各アルゴリズムの単精度浮動小数点数による畳み込みと平均値プーリングの実行時間と浮動小数点演算数を示している．最も実行時間は小さいものを太字にしている．cuDNN によるナイーブ手法では，Winograd アルゴリズムが採用されていた．浮動小数点数演算数は *nvprof -metrics flop_count_sp* で取得した．表 I における浮動小数点数演算数と *nvprof* による浮動小数点数演算数を比較すると，おおよそ一致していることがわかる．$n = 32, k = 3, c_i = 512, c_o = 512, p = 2$ の場合では，表 I から計算した Direct Sum 手法の浮動小数点数演算数は 1209925632，*nvprof* による Direct Sum 手法の浮動小数点数演算数は 1217946624 であった．cuDNN による naive 手法 (Winograd) の浮動小数点数演算数は，表 I の浮動小数点数演算数に比べ若干少ない箇所があるが，大きくても 3%程度なため，演算数は適切に表現できていると考えられる．Direct Sum 手法は多くのパラメータで，Fused Filter 手法や naive 手法より高速であった．これは Direct Sum 手法に必要な浮動小数点数演算数が少ないためである．また，cuBLAS による実装が cuDNN の実装に比べ高速である理由は，cuBLAS の行列積が極めて効率的であるためである．入力チャネル数及び出力チャネル数が 512 に場合に，cuDNN によるナイーブ手法 (Winograd) に比べ，Direct Sum 手法は約 1.8 倍の高速化を達成した．

Volta 世代の GPU である V100 は，半精度浮動小数点数の行列積を高速に行う演算器であるテンソルコアを搭載している．cuDNN や cuBLAS はテンソルコアを用いた計算をサポートしている．表 III は各アルゴリズムの半精度浮動小数点数による畳み込みと平均値プーリングの実行時間 [μs] と浮動小数点数演算数を示している．cuDNN による計算では，いくつかのパラメータ，手法において，テンソルコアが用いられない場合があった．そのため，表中の値は，テンソルコアを用いた計算がサポートされている場合は，テンソルコアを用いた場合と用いなかった場合で計算を行い，実行時間が小さいほうの実行時間と浮動小数点演算数を掲載している．テンソルコアを用いた計算がサポートされていない場合は，テンソルコアを用いずに計算した時間を掲載している．テンソルコアを用いた場合の計測結果を示す場合は，*を付加している．最も実行時間は小さいものを太字にしている．浮動小数点数演算数は *nvprof -metrics flop_count_hp* で取得した．半精度浮動小数点数の場合では，cuDNN は Winograd アルゴリズムではなく行列積関連のアルゴリズムが用いられていた．

はじめに，浮動小数点数演算数について，テンソルコアを用いずに計算したものは表 I とおおよそ一致している．しかしながら，テンソルコアを用いて計算したものは表 I よりもはるかに小さな値を示している．テンソルコアを用いた演算数は，上記のものでは取得できず，また，テンソルコアを用いた演算数を取得することはできない．つまり，減少分については，テンソルコアにより演算が行われたものと推測でき，実際の演算数は表 I と考えることができる．

実行時間を比較してみると，cuBLAS による Direct Sum 実装が最も高速であるとわかる．これは，必要な演算数が少なく，テンソルコアを用いた行列積が効率的に実装されているためであると推測できる．半精度浮動小数点数での畳み込みと平均値プーリングにおける計算は，$n = 32, k = 3, c_i = 512, c_o = 512, p = 2$ の場合に cuDNN によるナイーブ手法に比べ，4.4 倍の高速化を達成できた．

VI. まとめ

本論文では CNN における畳み込み層と平均値プーリング層における計算コストを軽減するアルゴリズムを 2 つ提案し，GPU に実装した．提案したアルゴリズムの 1 つは，平均値プーリングも畳み込み処理とみなして畳み込みカーネルとプーリングカーネルを先に畳み込んで Fused Filter を作成し，Fused Filter を用いて畳み込みとプーリングを行う Fused Filter 手法，もう 1 つは畳み込みとプーリングの計算順序を入れ替えることで計算コストを軽減する Direct Sum 手法である．NVIDIA Tesla V100 GPU 上に cuDNN，cuBLAS を用いて Fused Filter 手法と Direct Sum 手法を実装した結果，入力サイズ 32×32，畳み込みカーネルサイズ 3×3，プーリングサイズ 2×2，入力チャネル数 512，出力チャネル数 512 での単精度浮動小数点数による畳み込みと平均値プーリングの計算は cuDNN によるナイーブ手法に比べ，cuBLAS による Direct Sum 手法は 1.8 倍の高速化を達成した．また，半精度浮動小数点数での計算は，cuDNN によるナイーブ手法に比べ，cuBLAS による Direct Sum 手法は 4.4 倍の高速化を達成した．

REFERENCES

[1] Sharan Chetlur, Cliff Woolley, Philippe Vandermersch, Jonathan Cohen, John Tran, Bryan Catanzaro, and Evan Shelhamer. cudnn: Efficient primitives for deep learning. *arXiv preprint arXiv:1410.0759*, 2014.

[2] Mark Harris, Shubhabrata Sengupta, and John D. Owens. Chapter 8. summed-area variance shadow maps. In *GPU Gems 3*. Addison-Wesley, 2007.

[3] Akihiko Kasagi, Tsuguchika Tabaru, and Hirotaka Tamura. Fast algorithm using summed area tables with unified layer performing convolution and average pooling. In *Machine Learning for Signal Processing (MLSP), 2017 IEEE 27th International Workshop on*, pp. 1–6. IEEE, 2017.

[4] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. Imagenet classification with deep convolutional neural networks. In *Advances in neural information processing systems*, pp. 1097–1105, 2012.

TABLE II
NVIDIA V100 を用いた各手法の単精度浮動小数点数による実行時間 [μs] と浮動小数点数演算数

Input Channels		実行時間 [μs] Output Channels 32	64	128	256	512	浮動小数点数演算数 ($\times 10^5$) Output Channels 32	64	128	256	512
32	cuDNN (Naive, Winograd)	44	42	42	43	64	177	354	705	1410	2820
	cuDNN (Fused Filter)	35	33	33	36	39	84	168	336	672	1343
	cuDNN (Direct Sum)	70	70	70	70	70	99	99	194	383	762
	cuBLAS (Fused Filter)	35	33	35	34	41	85	170	338	676	1351
	cuBLAS (Direct Sum)	**34**	**32**	**30**	**33**	**35**	58	112	195	386	768
64	cuDNN (Naive, Winograd)	57	57	57	60	86	355	697	1400	2801	5566
	cuDNN (Fused Filter)	40	40	44	47	56	168	336	671	1343	2686
	cuDNN (Direct Sum)	102	102	102	102	102	197	197	387	765	1521
	cuBLAS (Fused Filter)	**35**	**33**	**35**	**34**	**41**	175	338	682	1364	2694
	cuBLAS (Direct Sum)	35	37	38	37	43	104	200	388	768	1528
128	cuDNN (Naive, Winograd)	88	88	88	104	133	694	1388	2773	5547	11094
	cuDNN (Fused Filter)	**43**	46	53	68	104	336	671	1343	2685	5370
	cuDNN (Direct Sum)	131	135	134	135	157	206	394	772	1527	3038
	cuBLAS (Fused Filter)	44	45	54	76	104	339	678	1353	2707	5413
	cuBLAS (Direct Sum)	44	**44**	**43**	**52**	**61**	208	397	783	1549	3046
256	cuDNN (Naive, Winograd)	147	146	153	175	223	1424	2769	5538	11076	22151
	cuDNN (Fused Filter)	405	60	83	145	174	671	1342	2685	5369	10739
	cuDNN (Direct Sum)	191	195	196	215	246	411	789	1544	3054	6075
	cuBLAS (Fused Filter)	411	62	83	146	181	718	1357	2714	5427	10854
	cuBLAS (Direct Sum)	**51**	**53**	**61**	**91**	**118**	425	817	1555	3076	6118
512	cuDNN (Naive, Winograd)	256	266	285	322	387	2797	5546	11030	22061	44121
	cuDNN (Fused Filter)	417	106	133	273	327	1342	2685	5369	10738	21476
	cuDNN (Direct Sum)	**399**	415	405	411	419	822	1577	3087	6108	12148
	cuBLAS (Fused Filter)	426	106	135	271	324	1389	2730	5398	10796	21591
	cuBLAS (Direct Sum)	421	**80**	**98**	**180**	**212**	1121	1644	3095	6123	12179

TABLE III
NVIDIA V100 を用いた各手法の半精度精度浮動小数点数による実行時間 [μs] と浮動小数点数演算数

Input Channels		実行時間 [μs] Output Channels 32	64	128	256	512	浮動小数点数演算数 ($\times 10^5$) Output Channels 32	64	128	256	512
32	cuDNN (Naive)	*49	*50	*50	*50	*54	*2	*3	*6	*11	*22
	cuDNN (Fused Filter)	194	190	200	200	198	336	336	336	672	1343
	cuDNN (Direct Sum)	*49	*49	*49	*49	*50	*9	*9	*10	*11	*13
	cuBLAS (Fused Filter)	*91	*83	*84	*84	*85	*1	*1	*1	*3	*5
	cuBLAS (Direct Sum)	***33**	***34**	***35**	***33**	***31**	*8	*8	*8	*9	*12
64	cuDNN (Naive)	133	133	138	137	136	1511	1511	1513	3025	6056
	cuDNN (Fused Filter)	283	284	291	291	294	671	671	671	1343	2686
	cuDNN (Direct Sum)	*70	*60	*60	*62	61	*18	*18	*18	*20	1531
	cuBLAS (Fused Filter)	*83	*88	*90	*96	*88	*2	*3	*3	*7	*5
	cuBLAS (Direct Sum)	***36**	***38**	***39**	***40**	***40**	*15	*15	*15	*16	*19
128	cuDNN (Naive)	*101	*81	*82	224	231	*4	*4	*7	6045	12090
	cuDNN (Fused Filter)	467	468	477	477	475	1342	1342	1343	2685	5370
	cuDNN (Direct Sum)	215	221	218	219	92	789	789	789	1544	3055
	cuBLAS (Fused Filter)	*104	*95	*96	*120	*117	*4	*4	*4	*9	*17
	cuBLAS (Direct Sum)	***48**	***48**	***48**	***61**	***59**	*30	*30	*30	*33	*32
256	cuDNN (Naive)	*164	*126	*126	412	413	*5	*5	*8	12085	24170
	cuDNN (Fused Filter)	839	837	854	856	856	2684	2685	2685	5369	10739
	cuDNN (Direct Sum)	405	415	408	408	218	1577	1577	1577	3088	6108
	cuBLAS (Fused Filter)	*102	*109	*113	*146	*153	*62	*15	*5	*9	*21
	cuBLAS (Direct Sum)	***55**	***61**	***60**	***98**	***99**	*58	*58	*58	*62	*71
512	cuDNN (Naive)	*206	*211	*202	774	830	*5	*9	*11	24164	48329
	cuDNN (Fused Filter)	1573	1578	1604	1606	1606	5369	5369	5369	10738	21476
	cuDNN (Direct Sum)	767	768	777	778	408	3154	3154	3154	6175	12215
	cuBLAS (Fused Filter)	*256	*148	*141	*231	*229	*62	*15	*5	*10	*21
	cuBLAS (Direct Sum)	***203**	***82**	***78**	***179**	***188**	*169	*122	*112	*118	*128

[5] Andrew Lavin and Scott Gray. Fast algorithms for convolutional neural networks. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pp. 4013–4021, 2016.

[6] Michael Mathieu, Mikael Henaff, and Yann LeCun. Fast training of convolutional networks through ffts. *arXiv preprint arXiv:1312.5851*, 2013.

[7] NVIDIA Corporation. *cuBLAS*. https://developer.nvidia.com/cublas.

[8] NVIDIA Corporation. *CUDA Zone*. https://developer.nvidia.com/cuda-zone.

[9] NVIDIA Corporation. *cuDNN Developer Guide :: Deep Learning SDK Documentation*. https://docs.nvidia.com/deeplearning/sdk/cudnn-developer-guide/index.html.

[10] NVIDIA Corporation. *How to Implement Performance Metrics in CUDA C/C++ — NVIDIA Developer Blog*. https://devblogs.nvidia.com/how-implement-performance-metrics-cuda-cc/.

[11] NVIDIA Corporation. *CUDA C Best Practice Guide Version 9.2*, 2018.

[12] NVIDIA Corporation. *CUDA C Programming Guide Version 9.2*, 2018.