7J-04

マルチGPU上での C++実装による畳み込みニューラルネットワークの並列処理

Parallelization of Convolutional Neural Network by C++ Implementation on Multi-GPU

綿貫 幸† 吉田 明正†
Yuki Watanuki Akimasa Yoshida

1 はじめに

畳み込みニューラルネットワーク（CNN）は，画像分類や物体検出などのディープラーニングに広く利用されているが，大量のデータによる学習が必要とされ，処理時間の短縮が課題となる．CNNで実行される行列演算の高速化には，GPUが活用されている．また，深層学習を並列処理により高速化・軽量化する手法として，モデル並列とデータ並列がある[1][2][3]．本稿では，C++実装によるCNNを用いた画像分類の学習を対象とし，データ並列を適用することで，マルチGPU環境での学習高速化を実現する．マルチGPU向け並列プログラムはCUDAとOpenMPを用いて作成されており，NVIDIA RTX A5000搭載サーバ上で行った性能評価の結果，提案手法の有効性が確認された．

2 CNNの並列処理

本章では，CNNの構成と，その並列処理手法であるデータ並列について述べる．

2.1 畳み込みニューラルネットワーク

ニューラルネットワークとは生物の神経回路を模したモデルであり，CNNは畳み込み層を持つニューラルネットワークである．本稿で扱うCNNを用いた画像分類プログラムはC++により実装されており，ネットワークの構成は表1の通りである[4]．畳み込み層はそれぞれ32枚のフィルタにより抽出された特徴マップを出力する．各フィルタは1つのフィルタ係数を重みパラメータとするユニットにより表される．畳み込み層及び全結合層の各ユニットは，ReLU活性化関数により発火を決定する．CNNの処理は，表1の第1層から第12層まで順伝播を行った後，逆伝播を行い，勾配によりパラメータを更新する．

2.2 データ並列

CNNのような深層学習に対し並列処理を行うアプローチとして，モデル並列とデータ並列がある．モデル並列は1つの学習モデルの処理を複数のノードで分割する手法であり，単一ノードに収まらない大規模なモデルの実装を可能にする．データ並列は入力データセットを分割し，複数のノードがそれぞれ複製された学習モデルの処理を行う手法である．ここで，CNNのミニバッチ学習は，入力データセットをバッチというグループに分け，バッチごとにパラメータを更新する．ミニバッチ学習における順伝播及び逆伝播の処理は各バッチ間で依存関係がないため，バッチ分割によるデータ並列を適用することができる．

†明治大学総合数理学部ネットワークデザイン学科
Department of Network Design, School of Interdisciplinary Mathematical Sciences, Meiji University

表1 CNNの構成．

層	層の種類	処理	出力サイズ
1	畳み込み層	3 × 3，32 フィルター	32 × 32
2	畳み込み層	3 × 3，32 フィルター	32 × 32
3	プーリング層	max プーリング	16 × 16
4	畳み込み層	3 × 3，32 フィルター	16 × 16
5	畳み込み層	3 × 3，32 フィルター	16 × 16
6	プーリング層	max プーリング	8 × 8
7	畳み込み層	3 × 3，32 フィルター	8 × 8
8	畳み込み層	3 × 3，32 フィルター	8 × 8
9	プーリング層	max プーリング	4 × 4
10	全結合層	512 ユニット	512
11	ドロップアウト層	確率 50%	512
12	全結合層	10 ユニット	10

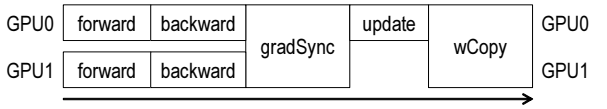

図1 マルチGPU上でのデータ並列CNNの処理手順．

3 マルチGPU環境でのCNNの並列処理

本稿では，CNNのミニバッチ学習にマルチGPUを利用し，データ並列を適用した高速化手法を提案する．

3.1 CNNへのデータ並列の適用

本稿で扱うCNNの各層はCUDAカーネルにより実装され，順伝播及び逆伝播の処理はGPU上で完結する．また，行列の四則演算にはNVIDIAが提供するcuBLASライブラリ関数を使用している．cuBLASでは行列をcolumn-majorとして取り扱うため，GPU上のデバイスメモリ上ではcolumn-majorとして行列を配置する．

2台のGPUによるデータ並列を適用したCNNの処理手順は，図1の通りである．まず，GPU0，GPU1は同一のモデルを持ち，それぞれ割り当てられた入力データに対し順伝播，逆伝播を行う．デバイス間で同期した後，GPU0は作成したgradSync()関数（図2）により全GPUの勾配gを集約し，平均値$\bar{g}$を算出する．最後に，$\bar{g}$により更新されたパラメータをwCopy()関数（図3）で全GPUに複製し，1バッチ分の学習が完了する．

3.2 OpenMPを用いたマルチGPU並列処理

マルチGPU上での並列処理の実装には，OpenMP[5]のsections構文を使用する（図4）．これにより，各sectionの処理が並列に実行された後，図1のgradSync，update，wCopyが実行される．

4 NVIDIA RTX A5000搭載サーバ上での性能評価

本章では，性能評価について述べる．

```
01: void gradSync(vector<Model> models){
    // 全GPU上のモデルを受け取る
02:     for(auto gs : models.at(0).graphs) { // 全層を読み込む
03:         if (全結合層 || 畳み込み層) {
04: /* GPU0 */  cudaSetDevice(0);
05:             cudaMemcpy(w, w0, size, cudaMemcpyDeviceToHost);
                // 更新済みパラメータをCPUホストメモリにコピー
06: /* GPU1 */  cudaSetDevice(1);
07:             cudaMemcpy(w1, w, size, cudaMemcpyHostToDevice);
                // 更新済みパラメータをCPUホストメモリからコピー;
08:         }
09:     }
10: }
```

図 2　gradSync() 関数.

```
01: void wCopy(vector<Model> models){
    // 全GPU上のモデルを受け取る
02:     for(auto gs : models.at(0).graphs) { // 全層を読み込む
03:         if (全結合層 || 畳み込み層) {
04: /* GPU0 */  cudaSetDevice(0);
05:             cudaMemcpy(w, w0, size, cudaMemcpyDeviceToHost);
                // 更新済みパラメータをCPUホストメモリにコピー
06: /* GPU1 */  cudaSetDevice(1);
07:             cudaMemcpy(w1, w, size, cudaMemcpyHostToDevice);
                // 更新済みパラメータをCPUホストメモリからコピー;
08:         }
09:     }
10: }
```

図 3　wCopy() 関数.

4.1　性能評価環境

性能評価に用いるマルチ GPU サーバの構成を，表 2 に示す．本性能評価では，CIFAR-10 画像データセット [6] を用いた CNN による画像分類プログラムに提案手法を適用し，学習時間を測定する．

表 2　性能評価に用いるマルチ GPU サーバの構成.

マシン	NVIDIA RTX A5000 搭載サーバ
プロセッサ	AMD EPYC 7443P (24Core, 2.85GHz, 128MB Cache)
メモリ	128GB
GPU	NVIDIA RTX A5000 x2
OS	Ubuntu 20.04 LTS
CUDA Toolkit	10.1
g++	8.4.0

4.2　マルチ GPU 環境でのデータ並列 CNN の性能評価

図 5 は，バッチサイズを 100 として CNN を用いた画像分類プログラムを逐次実行した場合と，提案手法により 2GPU で並列実行した場合の正解率を示す．図 5 の結果から，ともに 15 エポック程度で正解率 75%に達しており，提案手法の実装後もモデルが画像の分類精度を維持していることが確認された．

また，表 3 より，2GPU 実行時のエポック平均学習時間は 323.95[s] となり，逐次実行比で 1.69 倍の速度向上が得られた．なお，1GPU 実行時は逐次実行比 0.98 倍と低速化しており，gradSync() 関数，wCopy() 関数による GPU 間の転送処理のために実行時間が若干増加していることがわかる．以上の結果から，マルチ GPU 上での CNN による画像分類プログラムのデータ並列処理について，有効性が確認された．

5　おわりに

本稿では，マルチ GPU 環境における CNN の学習高速化のために，データ並列を適用する手法を提案した．マルチ GPU による並列処理は CUDA と OpenMP により実装している．マルチ GPU サーバ上での性能評価の結果，提案手法により逐次実行比 1.69 倍の速度向上が得られ，本手法の有効性が確認された．今後の課題としては，マルチ GPU 実行でボトルネックとなるデバイス間の通信時間の削減があげられる．

```
01: #pragma omp parallel sections private(gpu) num_threads(2)
02: {
03:     #pragma omp section
04:     {
05:         gpu = 0;
06:         cudaSetDevice(gpu); // GPU0に設定
07:         入力データをセット;
08:         順伝播;
09:         逆伝播;
10:     }
11:     #pragma omp section
12:     {
13:         gpu = 1;
14:         cudaSetDevice(gpu); // GPU1に設定
15:         入力データをセット;
16:         順伝播;
17:         逆伝播;
18:     }
19: }
```

図 4　OpenMP によるマルチ GPU 並列プログラム.

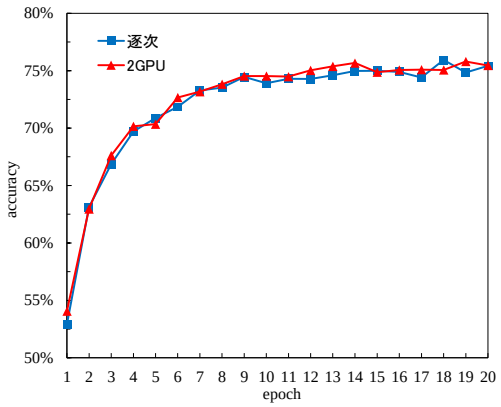

図 5　テストデータの正解率.

表 3　学習時間（エポック平均）.

	逐次	1GPU	2GPU
実行時間 [s]	545.90	558.40	323.95
速度向上率 [倍]	1.00	0.98	1.69

参考文献

[1] Chi-Chung Chen, Chia-Lin Yang, Hsiang-Yun Cheng. Efficient and Robust Parallel DNN Training through Model Parallelism on Multi-GPU Platform, arXiv:1809.02839v4, 2019.

[2] Ziqian Pei, Chensheng Li, Xiaowei Qin, Xiaohui Chen, Guo Wei. Iteration Time Prediction for CNN in Multi-GPU Platform: Modeling and Analysis, IEEE Access, Vol.7, pp.64788-64797, 2019.

[3] Lei Guan, Wotao Yin, Dongsheng Li, Xicheng Lu. XPipe: Efficient Pipeline Model Parallelism for Multi-GPU DNN Training, arXiv:1911.04610v3, 2020.

[4] 藤田毅. C++で学ぶディープラーニング, マイナビ出版, 2017.

[5] OpenMP. https://www.openmp.org/, 2019.

[6] The CIFAR-10 dataset. https://www.cs.toronto.edu/~kriz/cifar.html.