TensorFlow で大規模ニューラルネットワークを学習する手法の考察

今井 晴基　レドゥック トゥン　根岸 康　河内谷 清久二

ニューラルネットワークモデルの大規模化に伴い，学習時の GPU メモリーサイズの制約は大きくなってきている．それを回避する手法として，中間データを CPU に一旦退避し必要に応じて GPU に書き戻すことで GPU メモリーの効率的な利用を実現するデータ・スワッピング手法がある．しかし，この手法は CPU-GPU 間の通信による学習時間の増加をもたらす場合がある．もう一つの手法として中間データを一旦削除し，必要な時に再計算する再計算手法がある．この手法は追加の計算を必要とするが，CPU-GPU 間の通信は発生させない．本稿ではデータ・スワッピング手法に再計算手法を導入し CPU-GPU 間の通信を削減することで大規模深層学習の高速化を実現した．本手法を ResNet50 と DeepLabV3+に適用し，データ・スワッピング手法に対して，それぞれ最大 37.5% 18.7%の性能向上を実現した．

1　はじめに

深層学習は画像認識，音声認識など様々な分野において既存手法を圧倒する精度を達成し続けている．そのさらなる精度向上には，より深く大規模なニューラルネットワーク (NN) モデルが必要不可欠となってきており，大規模画像認識の競技会の ImageNet Large Scale Visual Recognition Challenge (ILSVRC) [15] においても，大規模な NN モデルを利用したチームが優勝してきた．また，実応用においても医療画像処理における 3 次元画像や高解像度の衛星画像の利用などにより，今後も NN モデルの大規模化は継続するものと考えられる．

NN モデルの大規模化に伴う演算量の増加などから，深層学習は GPU を用いて実行されることがほとんどである．GPU の演算能力は，新しいデバイスが出るたびに飛躍的に向上しているが，メモリー容量は，演算能力と比較して向上していない．GPU のメモリーは High Bandwidth Memory(HBM) など高速であるが高価なメモリーが搭載されていることなどから，今後もその飛躍的な増加は困難と考えられている．そのため，今後の大規模 NN モデルの実行には GPU メモリー容量の制約がより大きな課題となる．

その制約回避のために，メモリー圧縮手法なども開発されている [16] が，メモリー圧縮手法だけではさらなる大規模化に対処することには限界がある．NVIDIA® 社の CUDA® ライブラリが提供する機能 Unified Memory を利用すれば，アプリケーションが GPU の実メモリー以上のメモリーを利用可能になる．Unified Memory によりプログラムをほぼ変更することなく大規模な NN モデルの実行が可能になるはずであるが，GPU メモリーサイズを超えるデータを用いる場合に実用的な性能を実現するのは容易ではないことが報告されている [19] [20]．大規模な NN モデルを分割するモデル並列のアプローチも開発されてきているが [5] [8]，一般的に NN モデルは密に結合している部分も多いため，分散化した時の通信オーバーヘッドはより大きくなる可能性があることなどから広く利用されている段階にはない．

より大幅なメモリー削減を実用的な性能で実現す

* Consideration of training large-scale neural network models in TensorFlow.
This is an unrefereed paper. Copyrights belong to the Authors.

Haruki Imai, Tung Le Duc, Yasushi Negishi, Kiyokuni Kawachiya , 日本アイ・ビー・エム株式会社 東京基礎研究所, IBM Research - Tokyo.

る手法として，「データ・スワッピング手法」がある．データ・スワッピング手法は学習過程において生成される中間データ (特徴マップ) を一時的に CPU メモリーに退避し，利用するときに GPU へ書き戻すことにより GPU メモリーの効率的な利用を実現する．演算能力は GPU の方が圧倒的であるため，深層学習では CPU はあまり利用されておらず，CPU のメモリーも十分に活用されていないことが多い．このアプローチの課題は CPU-GPU 間の通信がボトルネックになってしまう可能性があることである．

一方，CPU-GPU 間の通信を必要としない手法として，「再計算手法」が存在する．この手法はデータ・スワッピング手法で CPU メモリーに退避する特徴マップを保存せずに一度削除してしまい，必要な時に再度計算する手法である．この手法の課題は追加の演算による性能低下と演算時の GPU メモリー使用量の増加である．

我々はデータ・スワッピング手法を TensorFlow で実現するソフトウェア「TensorFlow Large Model Support (TFLMS)」を開発した [6] [10] [12]．このソフトウェアは，TensorFlow の計算グラフを編集することでデータ・スワッピング手法を実現する．TFLMS には転送するメモリー量やタイミングを調整する性能最適化パラメータがあり，その自動チューニング機能を用いることで通信オーバーヘッドを削減することができる．

今回，我々は TFLMS を拡張し再計算手法を実現するグラフ編集を導入することで，データ・スワッピング手法と再計算手法をハイブリッドに利用する手法を実現した．本稿ではそれによる性能向上を報告する．以下，第 2 節では，計算グラフ編集による大規模 NN モデルの学習手法を記述し，第 3 節では，その TensorFlow における実装，第 4 節では実験と性能評価を行い，第 5 章で関連研究について記述し，第 6 節でまとめとする．

2 計算グラフ編集による大規模ニューラルネットワークモデルの学習手法

図 1 に通常の誤差逆伝播法の計算グラフ，データ・スワッピング手法のための編集後の計算グラフ，再計算手法のための編集後の計算グラフ，ハイブリッド手法の計算グラフを示す．計算グラフは説明のため簡略化したものを用いている．各図の実線はデータフロー (data dependency)，破線は制御フロー (control dependency) を示す．また，各手法の左側に順伝播 (FW)，右側は逆伝播 (BW) を示している．以下の節では，これを用いて各手法における計算グラフ編集について記述する．

2.1 データ・スワッピング手法

通常の誤差逆伝播手法においては順伝播で演算される中間データである特徴マップは，逆伝播における勾配の演算で用いられるため，その演算が完了するまで GPU メモリー上に保持される．特に畳み込みニューラルネットワークの場合，最初の層の特徴マップが大きい傾向があり，それらは逆伝播の後半まで保持されるため，多くの GPU メモリーを消費する．データ・スワッピング手法においては，その特徴マップを各層の計算後一時的に CPU メモリーに退避し，GPU で再度利用される際に GPU メモリーに書き戻すことで演算を行う．演算の際には必要な特徴マップのみ GPU に保持されるため，特に深い NN モデルの場合大幅な GPU メモリーの削減を実現可能である．CPU メモリーを大量に消費することになるが，GPU メモリーと比較すると安価であり容量も大きいため通常問題とならない．

計算グラフの編集においては，(1) 退避する特徴マップに対応するエッジを特定し，(2) そのエッジに退避 (スワップアウト) 用と書き戻し (スワップイン) 用のノードとして CPU で動作するノードを挿入する．(3) スワップインの開始のトリガーとなる制御フローを挿入する．演算は GPU で実行されるため，スワップアウトとスワップイン用のノードが実行される際に CPU-GPU 間の通信が発生する．スワップアウトは直前のノードの演算の完了後直ちに実施され，GPU に存在する特徴マップは GPU で不要となった時点で GPU メモリーからは解放される．スワップインは制御フローの依存関係にあるノードの実行後に実行される．その制御フローをより早く実行されるノードに挿入することよって通信と演算をオーバーラップ

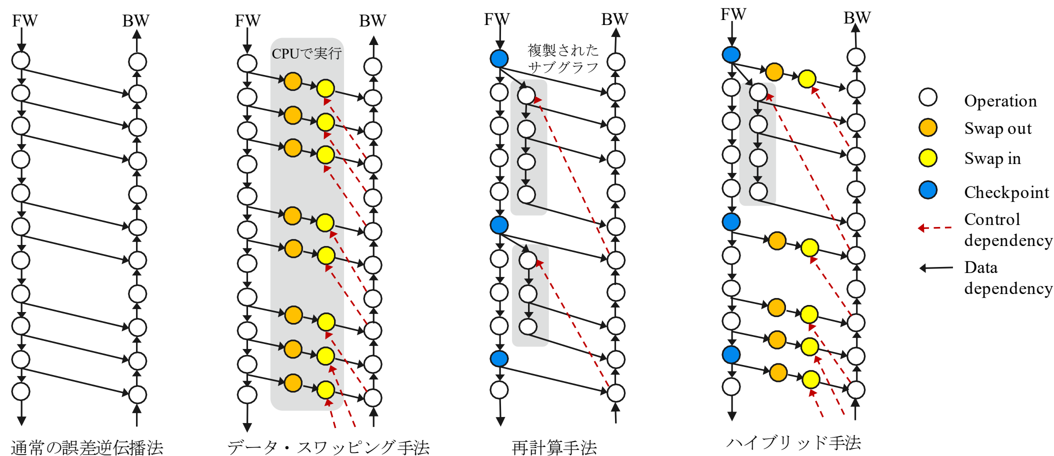

図 1　各手法のグラフ編集

させ，通信オーバーヘッドを削減することができる．

2.2　再計算手法

ここでは本稿で用いる再計算手法である Chen らの手法 [3] について述べる．再計算手法においては，順伝播で演算された特徴マップは順伝播のノードでの利用が終了した段階で解放される．全てのノードの特徴マップを解放しても順伝播の最初から再計算することによって再生成することはできるが，その場合，演算量の増加によるオーバーヘッドが大きくなる．そのため，特徴マップを一部解放せずに保存し，逆伝播ではその保存した特徴マップから再計算を実施することで演算の増加を削減する．その保存しておく特徴マップを checkpoint と呼ぶ．

計算グラフの編集においては，(1) 保存する checkpoint を決定し，(2) その checkpoint を生成するノード間のノードが構成するサブグラフを複製し，複製前のノードから逆伝播へのエッジを複製したサブブラフに付け替える．(3) 再計算開始のトリガーとして逆伝播において制御フローを挿入する．このようなグラフ編集を行うことで，順伝播では checkpoint 以外の特徴マップは順伝播での利用完了時に解放され，逆伝播では，再計算開始のトリガーにより checkpoint から必要な特徴マップが再計算されそれを利用し逆伝播の処理を継続する．どの checkpoint を残し，どのサブグラフを再計算対象とするかが最適化のポイントである．checkpoint としてそれを生成する計算量が多い特徴マップを選択し，その checkpoint の再計算を行わないことでオーバーヘッドを削減できる．そのため，畳み込み処理により生成された特徴マップを checkpoint とする手法がある [17]．本稿でもこの手法を利用する．

2.3　ハイブリッド手法

データ・スワッピング手法は CPU-GPU 間の新たな通信による性能悪化を招く可能性があるが，追加の演算は必要としない．一方，再計算手法では追加の演算が必要となるが，新たな通信は発生しない．そのため，データ・スワッピング手法において通信が性能悪化を招いていた場合，再計算手法を導入することで通信を削減し，性能向上を実現できる可能性がある．

計算グラフの編集においては，再計算させるサブグラフの決定とスワップアウト・インを挿入するエッジの決定を行い，それぞれにおいてグラフの編集を行う．

3　TensorFlow における実装

3.1　TensorFlow Large Model Support

我々はデータ・スワッピング手法を実現するソフトウェアの「TensorFlow Large Model Support

(TFLMS)] を開発した [6] [10] [12]．TensorFlow はユーザーが記述した NN モデルから計算グラフを作成し，セッションにおいてその計算グラフの学習を行う．セッション開始後は計算グラフは編集できないため，TFLMS はセッション開始前に，2.1 節に記述したグラフ編集を実施する．TFLMS には CPU にスワップする特徴マップの量やスワップインするタイミングを調整する性能最適化パラメータが導入されている．スワップアウトする特徴マップの量は，性能最適化パラメータの *swapout_threshold* によって調整可能である．TFLMS は各ノードのトポロジカルオーダーを用いてノード間の距離を計算し，その距離が *swapout_threshold* 以上のエッジにスワップアウト，スワップインのノードを挿入する．その距離が長いということは GPU メモリーに長い間その特徴マップが保存されていることになるため，その特徴マップをスワップアウトすることでより効率的に GPU メモリーを削減可能である．スワップインするタイミングは，*swapin_ahead* により調整可能である．*swapin_ahead* に大きな値を設定することでより早いノードにおいてスワップインが実行される．また，*swapin_groupby* は同じ特徴マップがスワップインされる場合にそれらを結合することでスワップインする特徴マップの量を削減するためのパラメータである．これらを適切に設定することで CPU-GPU 間の通信のオーバヘッドによる実行性能の低下を削減することができる．これらを調整するために自動チューニング機能が導入されており，ユーザープログラムに数行の編集を施すのみでグラフ編集は自動的に行われる．

3.2 ハイブリッド手法の導入

グラフ編集による再計算手法の実装として OpenAI が文献 [3] を元に開発した gradient-checkpointing がある [1]．これは TensorFlow における勾配の演算ルーチン tf.gradients() を置き換えることで 2.2 節のグラフ編集を実現している．この実装では，全てのサブグラフを再計算するようになっているため，ハイブリッド手法を行う際に必要な再計算させるサブグラフの選択を行うことはできない．そこで我々は TFLMS に独自に再計算手法を実現するグラフ編集を実装した．TFLMS に新たなパラメータ *checkpoint_list* を作成し，これに再計算対象とする checkpoint の ID を指定することでそこから構成されるサブグラフを再計算対象とする．checkpoint は，もう一つのパラメータ *recomp_checkpoint_ops_by_types* に checkpoint にするノードのタイプを指定する．本稿では Conv2D を checkpoint とした．ID はトポロジカルオーダー順に割り振られ，グラフを解析し ID と特徴マップ名を表示するようにしている．ユーザーはその ID を確認し指定するようにしている．ユーザーは通信プロファイルの調査や何度か試行することでより良い性能を示す ID を選択する必要がある．これらも自動設定が望ましいが，これは今後の課題である．

図 2 にハイブリッド手法におけるグラフ編集の流れを示す．TensorFlow が生成した計算グラフは，ユーザーパラメータ *checkpoint_list* で指定された checkpoint のサブグラフに対して再計算モジュールで再計算手法のためのグラフ編集を行う．そのグラフに対して自動パラメータ調整モジュールがデータ・スワッピング手法のパラメータを探索する．そのパラメータを元にデータ・スワッピングモジュールでデータ・スワッピング手法のためのグラフ編集を行う．*checkpoint_list* が指定されない場合，再計算モジュールでの編集は行われず，データ・スワッピング手法のみ適用される．また，再計算手法のみで十分 GPU メモリー使用量が削減できることが自動最適化モジュールで判断された場合，データ・スワッピングは不要と判断され，再計算手法のみが適用される．

4 実験と性能評価

4.1 実験環境

計測マシンには IBM® Power System™ AC922（以下，AC922）を用いた．AC922 は，CPU に POWER9 プロセッサー (10 コア，4GHz) を 2 基搭載し，CPU メモリー 512GB であり，2 ソケットの NUMA 構成となっている．NVIDIA® Tesla® V100(GPU メモリー 32GB) を 4 基搭載したモデルを利用した．計測にはそのうちの GPU1 基のみを用いた．CPU-GPU 間は NVLink2.0 で接続されており，4 基搭載モデルのため片方向 75GB/sec のバンド幅で

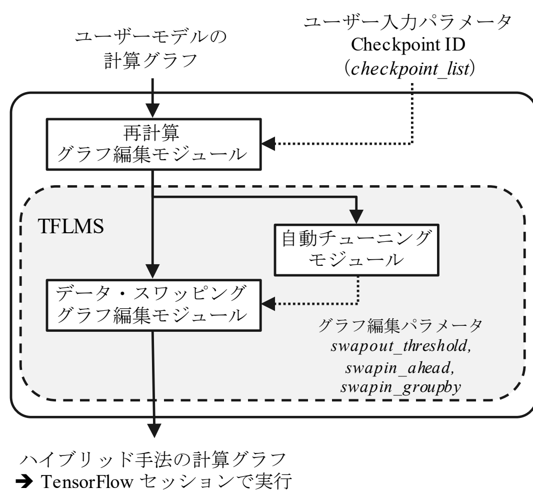

図 2　ハイブリッド手法におけるグラフ編集の流れ

接続されている．OS は Ubuntu 18.04.2 LTS, CUDA 10.1, cuDNN-7.5.1，TensorFlow 1.14.0rc0 を利用した．グラフ編集のため TFLMS version 2 [12] を用い、それに再計算手法を適用するための修正を行っている．

4.2　大規模ニューラルネットワークモデル

大規模 NN モデルとして Keras ResNet50 と DeepLabV3+ [2] を利用し評価を行う．Keras ResNet50 は [12]，DeepLabV3+は [11] において公開されている．通常，NN モデルを大規模化する際にバッチサイズを増加させることも多いが，大きなバッチサイズのモデルは複数の GPU を用いたデータ並列学習でも実現可能であるため，現在の GPU メモリーサイズで学習不可能なモデルとは言えない．そのため，本稿ではバッチサイズを固定とし画像サイズを拡大させたモデルを用いることで性能評価を行う．Keras ResNet50 はバッチサイズ 1，DeepLabV3+はバッチサイズ 16 を用いた．

4.3　学習可能な最大画像サイズ

図 3 に Keras ResNet50, 図 4 に DeepLabV3+の結果を示す．それぞれの図は横軸が画像サイズ，縦軸は実行性能として画像処理のスループットを示す．縦軸の単位はメガピクセル/秒である．横軸は画像の 1 辺のピクセス数であり，この値の 2 乗が画像 1 枚の総ピクセル数である．各図には手法を適用しない場合の TensorFlow(Vanilla TF)，データ・スワッピング手法 (Data swapping)，再計算手法 (Re-computation)，ハイブリッド手法 (Hybrid) が含まれている．Keras ResNet50 においては Vanilla TF の最大イメージサイズは 3000 であった．Re-computation はデータ・スワッピングは用いず全ての checkpoint を再計算している．この場合，最大画像サイズは 4500 まで増加することを確認した．Data swapping では最大画像サイズは 7000 であった．再計算手法においては checkpoint の特徴マップは GPU メモリーに保存されるが，データ・スワッピング手法では，全ての特徴マップをスワップすることができるため，再計算手法より大きな画像を学習可能である．Hybrid においても最大画像サイズは同じであったが，一般にハイブリッド手法では再計算手法を適用した部分のメモリーは増加するため，データ・スワッピングの方が最大画像サイズは大きくなる．データ・スワッピング手法のパラメータを手動でチューニングを行うと，データ・スワッピングの方がより大きな画像を学習できる可能性があると考えている．

図 4 の DeepLabV3+においても傾向は同様であり，Vanilla TF の最大画像サイズは 400 であった．Re-computation の最大画像サイズは 800，Data swapping，Hybrid は 1300 であった．

4.4　ハイブリッド手法による性能向上

次に実行性能について考察する．図 3 の Data swapping は，Vanilla TF の最大イメージサイズ 3000 以下では Vanilla TF と同じ性能を示している．この範囲では TFLMS の自動チューニング機能によりデータ・スワッピングは不要と判断されて，Vanilla TF と同じ実行となっているためである．Re-computation では自動チューニング機能はなく，常に再計算によるオーバーヘッドは発生するため，Vanilla TF と比較して性能は悪い．再計算による性能低下は Keras ResNet50 の画像サイズ 3000 において 15.8%であった．手動で再計算する checkpoint を選択することでこの性能低下は小さくなると考えられる．縦軸のスループットは画像サイズが大きくなるに従い GPU の演算効率の

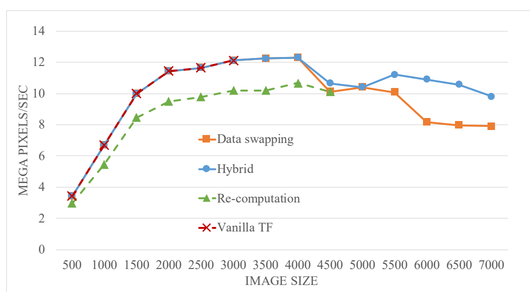

図 3　Keras ResNet50 の性能

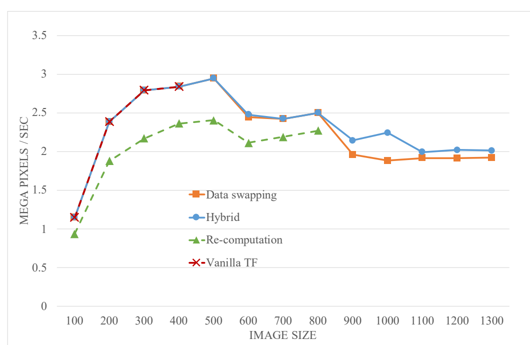

図 4　DeepLabV3+の性能

向上により上昇するが，それ以降はある一定値を示す (Keras ResNet50 の場合は 12 メガピクセル/秒程度)．一定値を示している場合は，各手法のオーバーヘッドはないことを示す．Data swapping では画像サイズ 4000 までは一定であり性能悪化を示さなかった．これは TFLMS が適切なパラメータを選択し通信を計算とオーバーラップさせることができたためと考えられる．Data swapping では画像サイズ 4500 以上については徐々に性能は悪化している．これはスワップを必要とする特徴マップ数の増加により，完全には計算とオーバラップをすることができなくなったためと考えられる．図 4 の DeepLabV3+でも傾向は同様であり，画像サイズ 400 以下では Data swapping と Vanilla TF は同じ性能を示している．画像サイズ 400 における再計算による性能低下は 16.7%であった．Data swapping は画像サイズ 500 までは性能悪化を示していない．表 1 表 2 に TFLMS の自動チューニングモジュールで選択されたパラメータを示す．データスワピング手法 (Data swapping) においては，画像サイズの増加に伴い *swapout_threshold* の値を小さくすることでより多くの特徴マップをスワップするようなパラメータが選択されている．*swapin_ahead* に関しては，小さい画像サイズでは GPU メモリーに余裕があるため値を大きくし早めのスワップインが可能であるが，画像サイズの増加に伴い，それが不可能になり小さな値を選択している．

ハイブリッド手法 (Hybrid) においては，図 3 図 4 に示すように Data swapping と比べて Keras ResNet50 で最大 37.5%(画像サイズ 6500)，DeepLabV3+で 18.7%(画像サイズ 1000) と性能向上している．再計算のために選択した checkpoint は表 1 と表 2 に示している．これは checkpoint の組み合わせをいくつか試行し良い性能を示した設定である．図 2 に示したように，再計算のための編集後のグラフに対して自動チューニングが実施されているため，TFLMS のパラメータは Data swapping と異なる．Data swapping と比較して *swapout_threshold* に変化はほぼ見られなかったが，*swapin_ahead* は大きくすることができている．このパラメータを利用したグラフ編集において実際に挿入されたスワップアウトのノード数を図 5 と図 6 に示す．Hybrid においてスワップアウトされる特徴マップ数は，Data swapping と比べ減少していることがわかる．これにより通信が削減され性能が向上したと考えられる．今回は再計算手法のパラメータは手動で選択したため，より良い組み合わせがある可能性がある．データ・スワッピング手法のパラメータ同様の自動チューニング手法も開発中である．

5　関連研究

データ・スワッピング手法としては，様々な手法が提案されている [4] [10] [13] [14] [18]．その中で計算グラフを用いているものは，本稿で用いた TFLMS [10] と Meng ら [13] による TensorFlow の計算グラフを編集する手法である．また，再計算手法としては Chen らの [3] や Kusumoto らの [9] がある．本稿では Chen らのグラフ編集を用いている．

ハイブリッド手法としては，Wang ら [17] や Ito ら [7] の報告がある．Wang らは畳み込み層をスワップアウトの対象とし，それ以外を再計算対象としてい

表 1　TFLMS の自動チューニングモジュールで選択されたパラメータと再計算手法のパラメータ (Keras ResNet50)

	Image size	3500	4000	4500	5000	5500	6000	6500	7000
Data swapping	*swapout_threshold*	743	639	593	529	501	463	429	399
	swapin_ahead	217	74	49	30	28	23	17	9
	swapin_groupby	802	802	802	802	802	802	802	802
	checkpoint_list								
Hybrid	*swapout_threshold*	743	639	593	529	477	463	429	399
	swapin_ahead	217	74	68	34	25	30	25	9
	swapin_groupby	802	802	817	808	808	821	821	808
	checkpoint_list			2,4,6	2	2	2,4,6,7	2,4,5,6	2

表 2　TFLMS の自動チューニングモジュールで選択されたパラメータと再計算手法のパラメータ (DeepLabV3+)

	Image size	500	600	700	800	900	1000	1100	1200	1300
Data swapping	*swapout_threshold*	1427	1236	650	456	217	174	161	121	111
	swapin_ahead	906	381	77	30	11	12	6	3	3
	swapin_groupby	1501	1501	1501	1501	0	22	1501	1501	22
	checkpoint_list									
Hybrid	*swapout_threshold*	1427	1236	650	456	217	174	164	121	111
	swapin_ahead	906	575	77	30	30	31	6	3	33
	swapin_groupby	1501	1508	1501	1501	1508	1525	1539	1501	23
	checkpoint_list		1,2,3,4			2,3,4	1,2,3,4,5,6	8,9,11.12,13,14,15,16,17,18	8,9	8,9,11,12,13,14,15,16,17,18

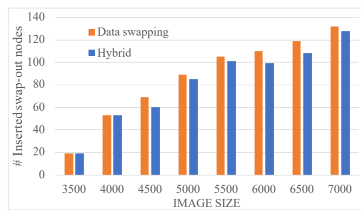

図 5　Keras ResNet50 におけるグラフ編集によって挿入されたスワップアウトノード数

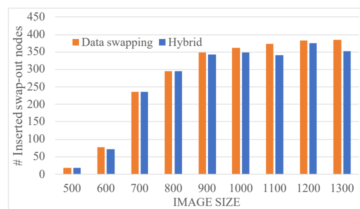

図 6　DeepLabV3+におけるグラフ編集によって挿入されたスワップアウトノード数

る点では我々の手法と同等である．スワップインのタイミングを直前の畳み込み層にするなどスワップアウト・インのパラメータは固定にしているが，我々は TFLMS の自動チューニングを用いてより柔軟に決めている点が異なる．Ito らは事前の実行プロファイルに基づきデータ・スワッピング部分と再計算部分を決定することで Wang らより良い性能を報告している．これらの手法においては計算グラフを用いておらず，本稿とは異なる．計算グラフを用いることで NN モデルの全体を考慮した最適化が可能になることで，今後より良い性能を実現できると考えている．

6　まとめと今後の課題

本稿では，GPU メモリーサイズを超えるような大規模な NN モデルを実行する手法であるデータ・スワッピング手法に，再計算手法を導入することでデータ・スワッピング手法の課題である CPU-GPU 間の通信を削減する手法を提案した．TensorFlow の計算グラフを編集しデータ・スワッピング手法を実現する TFLMS に再計算手法のグラフ編集機能を導入することで，2 つのハイブリッド手法を実現した．本手法を ResNet50 と DeepLabV3+に適用することで，データ・スワッピング手法に対して，それぞれ最大 37.5% 18.7%の性能向上を実現した．データ・スワッピングのグラフ編集の性能最適化パラメータは自動チューニングが実施されるが，再計算手法のパラメータは手動

でのチューニングが必要である．その自動チューニングが今後の課題である．

参考文献

[1] Bulatov, Y.: Fitting larger networks into memory, https://medium.com/tensorflow/fitting-larger-networks-into-memory-583e3c758ff9. Accessed: 2019-08-03.

[2] Chen, L.-C., Zhu, Y., Papandreou, G., Schroff, F., and Adam, H.: Encoder-Decoder with Atrous Separable Convolution for Semantic Image Segmentation, *ECCV*, 2018.

[3] Chen, T., Xu, B., Zhang, C., and Guestrin, C.: Training deep nets with sublinear memory cost, *arXiv preprint arXiv:1604.06174*, (2016).

[4] Cho, M., Le, T., Finkler, U., Imai, H., Negishi, Y., Sekiyama, T., Vinod, S., Zolotov, V., Kawachiya, K., Kung, D. S., et al.: Large model support for deep learning in caffe and chainer, *SysML*, (2018).

[5] Dean, J., Corrado, G., Monga, R., Chen, K., Devin, M., Mao, M., Senior, A., Tucker, P., Yang, K., Le, Q. V., et al.: Large scale distributed deep networks, *Advances in neural information processing systems*, 2012, pp. 1223–1231.

[6] Imai, H., Matzek, S., Le, T. D., Negishi, Y., and Kawachiya, K.: Fast and Accurate 3D Medical Image Segmentation with Data-swapping Method, *arXiv preprint arXiv:1812.07816*, (2018).

[7] Ito, Y., Imai, H., Duc, T. L., Negishi, Y., Kawachiya, K., Matsumiya, R., and Endo, T.: Profiling based out-of-core hybrid method for large neural networks: poster, *Proceedings of the 24th Symposium on Principles and Practice of Parallel Programming*, ACM, 2019, pp. 399–400.

[8] Krizhevsky, A.: One weird trick for parallelizing convolutional neural networks, *arXiv preprint arXiv:1404.5997*, (2014).

[9] Kusumoto, M., Inoue, T., Watanabe, G., Akiba, T., and Koyama, M.: A Graph Theoretic Framework of Recomputation Algorithms for Memory-Efficient Backpropagation, *arXiv preprint arXiv:1905.11722*, (2019).

[10] Le, T. D., Imai, H., Negishi, Y., and Kawachiya, K.: Automatic GPU memory management for large neural models in TensorFlow, *Proceedings of the 2019 ACM SIGPLAN International Symposium on Memory Management*, ACM, 2019, pp. 1–13.

[11] Liang-Chieh Chen, YuKun Zhu, G. P. H. H. M. D. C. T. L.: DeepLab: Deep Labelling for Semantic Image Segmentation, https://github.com/tensorflow/models/tree/master/research/deeplab. Accessed: 2019-08-03.

[12] Matzek, S.: TensorFlow Large Model Support Resources, https://developer.ibm.com/linuxonpower2019/06/11/tensorflow-large-model-support-resources/. Accessed: 2019-08-03.

[13] Meng, C., Sun, M., Yang, J., Qiu, M., and Gu, Y.: Training deeper models by GPU memory optimization on TensorFlow, *Proc. of ML Systems Workshop in NIPS*, 2017.

[14] Rhu, M., Gimelshein, N., Clemons, J., Zulfiqar, A., and Keckler, S. W.: vDNN: Virtualized deep neural networks for scalable, memory-efficient neural network design, *The 49th Annual IEEE/ACM International Symposium on Microarchitecture*, IEEE Press, 2016, pp. 18.

[15] Russakovsky, O., Deng, J., Su, H., Krause, J., Satheesh, S., Ma, S., Huang, Z., Karpathy, A., Khosla, A., Bernstein, M., et al.: Imagenet large scale visual recognition challenge, *International journal of computer vision*, Vol. 115, No. 3(2015), pp. 211–252.

[16] Shirahata, K., Tomita, Y., and Ike, A.: Memory reduction method for deep neural network training, *2016 IEEE 26th International Workshop on Machine Learning for Signal Processing (MLSP)*, IEEE, 2016, pp. 1–6.

[17] Wang, L., Ye, J., Zhao, Y., Wu, W., Li, A., Song, S. L., Xu, Z., and Kraska, T.: Superneurons: dynamic GPU memory management for training deep neural networks, *ACM SIGPLAN Notices*, Vol. 53, No. 1, ACM, 2018, pp. 41–53.

[18] 今井晴基, 根岸康, 関山太朗, 河内谷清久仁, ほか: 大規模ニューラルネットワークモデルの Out-of-Core 学習の性能評価, 研究報告ハイパフォーマンスコンピューティング *(HPC)*, Vol. 2017, No. 25(2017), pp. 1–7.

[19] 根岸康, 今井晴基, 土井淳, 河内谷清久仁: Unified Memory を用いた大規模ディープラーニングモデルの性能に関する考察, 日本ソフトウェア科学会大会論文集, Vol. 34(2017), pp. 11–21.

[20] 根岸康, 今井晴基, 土井淳, 河内谷清久仁: CUDA Unified Memory のディープラーニングへの適用についての考察, 研究報告ハイパフォーマンスコンピューティング *(HPC)*, Vol. 2018, No. 7(2018), pp. 1–8.