Vol.2018-HPC-165 No.30
2018/8/1

GPUメモリ管理の実行時最適化による大規模深層学習の高速化

伊藤　祐貴[1,a)]　今井　晴基[2]　レドゥック トゥン[2]　根岸　康[2]　河内谷　清久仁[2]　松宮　遼[1]
遠藤　敏夫[1]

概要：ニューラルネットワーク (NN) による深層学習は計算量が非常に大きく，その計算は GPU を用いることで汎用 CPU を用いた場合よりも高速に行われることが示されている．しかし，GPU メモリ容量が小さいために，問題サイズが大きい NN を GPU で高速に計算するプログラムを実装することは困難である．GPU のメモリ容量を超える問題サイズの NN を計算するための手法としては, data-swapping 手法と recompute 手法が提案されている．しかし, これらの手法ではデータ移動や計算量増加によるオーバーヘッドが発生してしまう．それらのオーバーヘッドを削減するためには, どのデータを swap 対象または recompute 対象とするかを適切に選択することが重要な課題となる．この課題に対して, 本研究では swap 対象と recompute 対象を実行時プロファイリングに基づいて最適化する手法を提案した．また, 我々は提案手法を深層学習フレームワーク Chainer を拡張することで実装し, 性能評価を行った．その結果, 50 GB 以上のメモリを必要とする NN の計算がメモリ容量 16 GB の GPU 一台で可能となり, その際の性能低下は 34 %であった．

1. はじめに

近年，画像認識や画像生成, 音声認識，自然言語処理などの分野にニューラルネット (NN) を適用することにより，高い精度が示されている [1][2][3][4]．ニューラルネットワークとは人間の脳の学習メカニズムを基にした計算モデルである．

NN による深層学習は計算量が多いが，GPU を用いることで高速に行うことができる． しかし，GPU を使用して計算できる NN の大きさは GPU のメモリ容量によって制限されてしまう． 例として，バッチサイズを 640 とした場合の ResNet50[5] では 50 GB 以上のメモリを必要とする．これに対して NVIDIA の Tesla V100 のメモリ容量は高々 32 GB しかない．そのため，Tesla V100 一台ではこの NN の計算を GPU を用いて行うことは困難である．

この課題を解決するための手法としては data-swapping 手法 [6][7][8] と recompute 手法 [9] の 2 つが提案されている．これらの手法により大規模な NN を GPU を用いて計算することは可能となる．しかし, data-swapping 手法と recompute 手法ではそれぞれ CPU-GPU 間通信や計算量の増加によって性能へのオーバーヘッドが生じてしまう．そして, それらのオーバーヘッドを削減するためには, どのデータを swap 対象または recompute 対象とするかが課題となる．一方, NN の計算における計算時間やメモリ使用量は, NN の構造や実行環境に大きく依存している．そのため, 最適な swap 対象と recompute 対象を静的に決定するのは難しい．

この課題に対して, 本論文では data-swapping 手法と recompute 手法を用いた NN の計算を高速化するための最適化手法について述べる．提案手法では実行時プロファイリングに基づいて swap 対象および recompute 対象を最適化することにより, 性能オーバーヘッドを削減する．また, 各 swap 通信の scheduling による高速化も行う．

我々は提案手法を深層学習フレームワーク Chainer[10] を拡張することで実装した．そして, 提案手法により, 50 GB 以上のメモリを必要とする NN の計算がメモリ容量 16 GB の GPU 一台で可能となり, その際の性能低下を 34% に抑えることができた．

2. 背景

2.1 ニューラルネットワーク

2.1.1 構造

NN の構造の概念図を図 1 に示す．NN は複数の層から

[1] 東京工業大学
[2] 日本アイ・ビー・エム株式会社 東京基礎研究所
[a)] itou.y.aj@m.titech.ac.jp

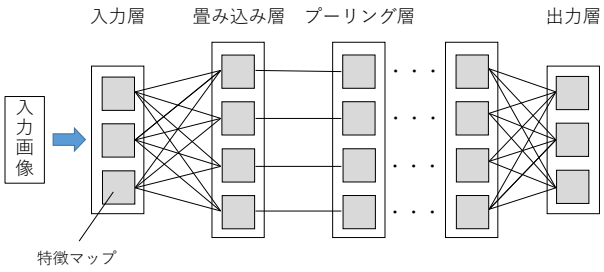

図 1: NN の構造

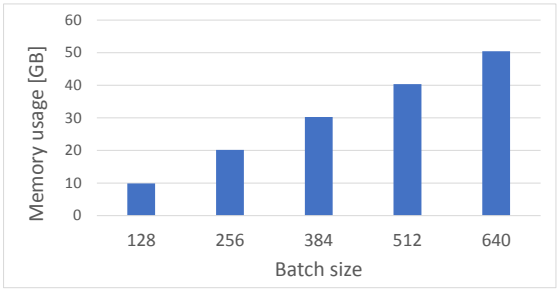

図 3: ResNet50 のメモリ使用量

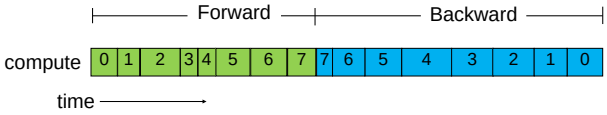

図 2: NN の計算処理のタイムラインの例 (層数=8)

構成され，各層はさらに複数の特徴マップから構成されている．特徴マップとは前の層の出力を入力として受け取り，計算を行うことで出力されるデータである．

層の種類は計算の内容によって畳み込み層やプーリング層, Batch-Normalization(BN) 層, 全結合層などに分類される．また, 畳み込み層や全結合層では計算のために重みフィルタなどのパラメータを必要とする．そして, NN の学習ではそれらのパラメータが最適化対象となる．

2.1.2 forward と backward

NN の学習は誤差逆伝播法と呼ばれる手法を用いて行われる．誤差逆伝播法とは NN の出力と正解データの誤差を用いてパラメータの更新を行う学習法である．そのため, 誤差逆伝播法では以下の 3 つの処理を行う．

(1) 学習のためのサンプルデータをネットワークの入力として, 各層の特徴マップを計算する [forward]
(2) ネットワークの出力と正解データの誤差を用いて, 各層の特徴マップや重みフィルタなどに対する勾配を求める [backward]
(3) 計算した勾配を用いて, パラメータを更新する

ここで, 各層の特徴マップは前の層の特徴マップから計算されるため, forward の計算は入力層から出力層へと 1 層ずつ行われる．一方, 各層における勾配は後の層の勾配から計算されるため, backward の計算は出力層から入力層へと順に行われる．そのため, NN の計算全体のタイムラインは図 2 のようになる．図 2 において, 緑と水色の四角はそれぞれ各層の forward または backward 計算の実行時間を表す (内部の数字は forward での計算順に各層に割り振った番号とする)．

また, 一般的に backward における勾配の計算では forward で計算した特徴マップも入力として用いる．よって, forward で一度計算した特徴マップのデータは backward で使用されるまでメモリ上に保持される必要がある．

2.1.3 バッチ処理

画像 1 枚を入力として NN の計算を GPU で行う場合，小規模な NN では並列度が低いため，GPU の演算性能を活かせないことがある．そのため，複数の入力画像をまとめて NN への入力とするバッチ処理を行う．ひとまとめにした画像の数をバッチサイズと呼ぶ．

2.2 NN のメモリ使用量

2.1.2 節で述べたように, 誤差逆伝播法では backward で使用するために層ごとに forward で計算した特徴マップを保持する必要がある．また, 特徴マップ以外にも重みフィルタなどのパラメータおよび勾配のためにメモリを使用する．これらに加えて, 深層学習プログラムの実装によっては計算高速化のために workspace を必要とすることがある．

NN の計算において, 層あたりのメモリ使用量はバッチサイズや特徴マップの大きさに比例して増加する．また, NN 全体のメモリ使用量は層数によっても変化する．そのため, バッチサイズや入力データが大きい NN の計算を行うためには多くのメモリが必要となる．

例として，ResNet50 ネットワークの計算に必要なメモリ量を図 3 に示す．図 3 ではバッチサイズが 640 の場合にメモリ使用量が 50 GB を超えている．一方，Tesla V100 のメモリ容量は高々 32 GB しかない．これは NN の計算において GPU メモリが足りなくなる場合があることを表している．

3. GPU メモリ容量を超える深層学習手法

深層学習において GPU メモリ容量を超えるデータを扱うための手法としては data-swapping 手法と recompute 手法の 2 つが提案されており, 本章ではこれらの手法について述べる．

3.1 data-swapping 手法

data-swapping 手法では, まず各層の forward 計算で使用されたデータの一部を CPU メモリに swap-out する．swap-out の例を図 4 に示す．図 4 のように, データ X を入力としてデータ Y を出力する forward 計算を行った場合, その計

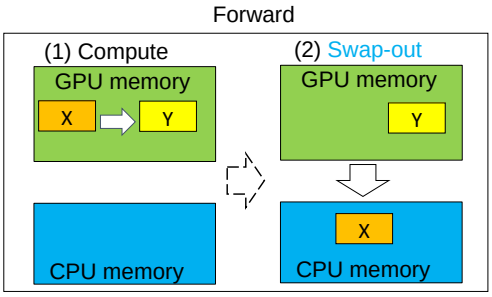

図 4: data-swapping 手法における forward 時の swap-out

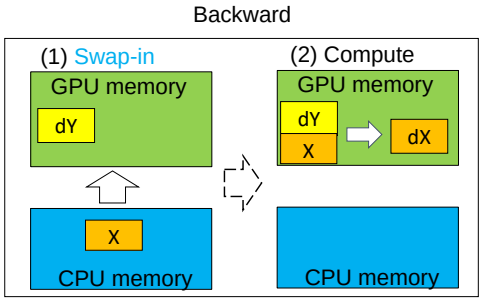

図 5: data-swapping 手法における backward 時の swap-in
(dX と dY はそれぞれ X・Y に対応する勾配)

算終了後に X を CPU メモリへと swap-out する．そして，forward であるデータを swap-out した場合，backward 時にそのデータを swap-in する必要がある．swap-in の例を図 5 に示す．図 5 では swap-out されたデータ X が backward 計算に使用される場合，先に X を GPU メモリへと swap-in する．そして，その後 X を使用して計算が行われる．

また，一般的に CPU-GPU 間通信によるオーバーヘッドを削減するために，「forward 計算と swap-out 通信」および「backward 計算と swap-in 通信」はそれぞれパイプライン処理される．

性能への影響

以上のように，data-swapping 手法ではデータの swap のために追加の CPU-GPU 間通信が行われる．次にそれらの通信による性能への影響について述べる．

まず，あるデータを swap-out するためには，そのデータを使用する forward 計算がすべて終了するのを待たなければならない．同様に，各層の backward 計算を行うためには，その計算に使用されるすべてのデータが swap-in されるのを待たなければならない．このように各層の計算と swap には依存関係がある．これらに加えて，backward 計算を開始するためには，forward における swap-out がすべて終了するのを待つ必要がある．

data-swapping を行う場合の NN の計算処理のタイムラインの例を図 6 に示す．図 6 のように各 swap-out は対応する forward 計算の終了後に開始する．また，各 backward 計算は対応する swap-in の終了後に開始する．そして，各 swap の終了を待つために計算を行うことができない時間 (図 6 の赤枠部分) が data-swapping によるオーバーヘッドである．一方，パイプライン処理により計算で通信を隠蔽できる場合は通信によるオーバーヘッドを削減することができる．

3.2 recompute 手法

recompute 手法では，forward 時に計算および使用された特徴マップのデータの一部を GPU メモリ上で保持せずに解放する．そして，backward 計算を行う際に必要な特徴マップがメモリ上に保持されていなかった場合，再度 forward の計算を行い特徴マップのデータを用意する．
recompute 手法の例を図 7 と図 8 に示す．forward において図 7 のように X から Y，Y から Z を計算した場合について考える．このとき Y が recompute 対象ならば Y のデータを GPU メモリ上から解放する．そして，backward において Y が必要になった際には図 8 のように，再度 X から Y を計算してから backward 計算を行う．

3.3 hybrid 手法

以上のように data-swapping 手法および recompute 手法を用いることで，GPU メモリ容量を超える問題サイズの NN の計算は可能となる．しかし，data-swapping 手法では CPU-GPU 間通信，また recompute 手法では再計算によって実行時間へのオーバーヘッドが発生する．ここで，この 2 手法を用いた際のオーバーヘッドにはそれぞれ以下のような特徴がある．

- data-swapping 手法： 計算量の大きい層 (畳み込み層など) が多い NN では swap を行っても，計算によって通信オーバーヘッドを隠蔽しやすい．これに対して，計算量の小さい層 (BN 層など) が多い NN の場合は通信を隠蔽しにくい．
- recompute 手法： 計算量の大きい層の特徴マップを recompute 対象とするとオーバーヘッドが大きくなってしまうが，計算量の小さい層を recompute 対象とする場合はオーバーヘッドは小さい．

これらの特徴を考慮して，Wang らは性能向上のために data-swapping と recompute の両方を組み合わせることを提案している [11]．以下，本論文では data-swapping と recompute の両方を用いる手法を hybrid 手法と呼ぶ．

Wang らの手法では swap 対象や recompute 対象を静的に決定しているが，これは実際に計算する NN の各計算の実行時間やメモリ使用量を詳細に考慮していない．一方で NN の各層の計算量やメモリ使用量は NN の構造によって異なる．そのため，NN 毎の特徴を考慮して swap 対象や recompute 対象を決定することにより，静的に決定した場合より性能が向上することが期待される．

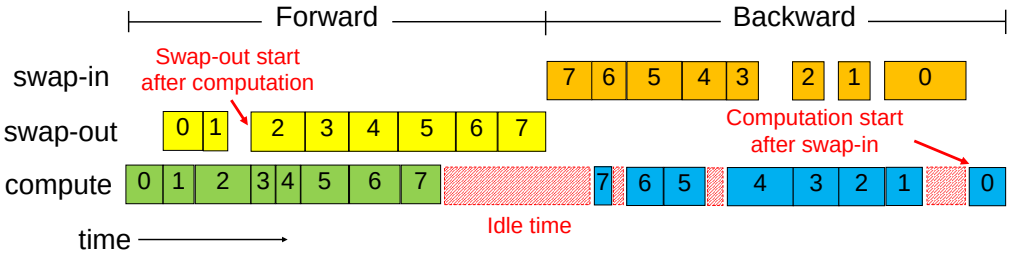

図 6: data-swapping を行う場合の NN の計算処理のタイムラインの例 (層数=8). 黄色とオレンジの四角は各層の計算に対応する swap の実行時間を表す. この例ではすべての層においてデータの swap を行い, かつ各 swap-in は一つ後ろの層の計算と同時に開始している.

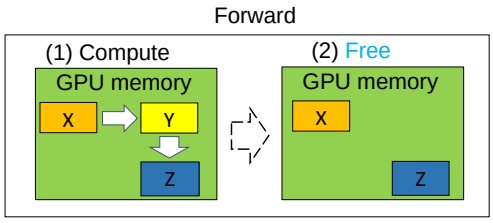

図 7: recompute 手法における forward 時のメモリ解放

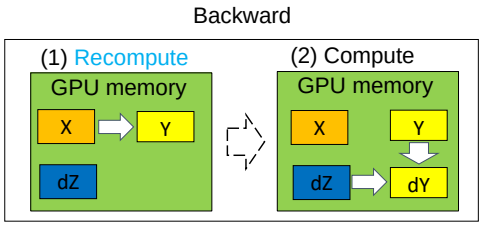

図 8: recompute 手法における backward 時の再計算
(dY と dZ はそれぞれ Y・Z に対応する勾配)

4. hybrid 手法の実行時最適化

4.1 最適化の概要

4.1.1 最適化対象

提案手法では NN の計算で使用される各データに以下のいずれかの属性をそれぞれ割り当てることによって, 大規模な NN の計算を高速化する.

- “keep”：GPU メモリ上で保持する
- “swap”：swap 対象とする
- “recompute”：recompute 対象とする

ここで, 現在の設計ではこの属性割り当ては各層の特徴マップに対してのみ行うものとしている. そして, 特徴マップ以外のデータについてはすべて GPU メモリ上で保持することとする. これは重みフィルタなどのパラメータは特徴マップと比較して小さく, また各勾配は GPU メモリ上で保持される期間が短いためである.

上記の属性割り当てに加えて, 提案手法では各 swap-in の実行タイミングの最適化も行う.

4.1.2 実行時間とメモリ使用量の予測

この最適化の目的は「NN の計算全体の実行時間」を小さくすることである. また, メモリ使用量に関して「処理のいずれの時点においても GPU メモリ使用量が GPU メモリ容量を超えない」という条件を満たさなければならない. しかし, 計算と通信のパイプライン処理や同期があるために, 実行時間を簡単な線形方程式などで定式化することは困難である. 同様にメモリ使用量についても swap のタイミングなどで malloc/free の順番が変わるために定式化は難しい.

そのため, 最適化のための探索の際には, 各特徴マップへ属性をそれぞれ割り当てた場合の実行タイムラインおよびメモリ管理をシミュレートし, 処理全体の実行時間とメモリ使用量を予測することが必要となる.

4.1.3 最適化全体の流れ

4.2 節で後述するが, 現在の設計では上記の実行時間とメモリ使用量の予測のために実行時 profiling を必要とする. そして, その profiling 結果に基づいて最適化が行われる. よって, 本手法を用いる場合の深層学習は以下のような流れで実行される.

(1) 学習イテレーションを数回実行して各メモリ管理・実行時間を記録 (profiling)
(2) 属性割り当ての決定, および各 swap-in の実行タイミングの最適化
(3) 最適化の結果を用いて学習イテレーションを続ける

以下, 4.2 節では実行時 profiling, 4.3 節で swap-in の実行タイミングの最適化, 4.4 節では最適化のための探索についてそれぞれ説明する.

4.2 実行時 profiling

本節では提案手法において実行時 profiling を行う理由について述べる. まず, 4.1.2 項の実行時間とメモリ使用量の予測には NN の計算における以下の情報が必要である.

- 各 malloc/free のサイズと順番
- 各 forward/backward 計算の実行時間
- 各 swap-out/swap-in 通信の実行時間
- 各 swap-out/swap-in の実行タイミング
- 計算グラフおよび各データの依存関係

しかし, これらの情報をすべて静的に予測するのは困難である. たとえば, 各 swap の実行タイミングは NN の構造に

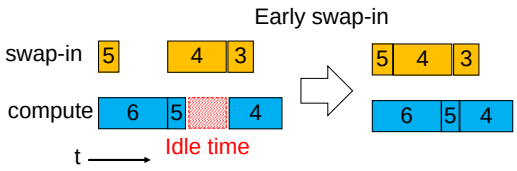

図 9: swap-in の scheduling の例

依存しており, 分岐などが多い複雑な NN では予測が困難である. その他のメモリ管理や計算の順序についても深層学習プログラムの実装に大きく依存しており, 各 malloc や free のサイズと順番なども含めて, すべての情報を静的に予測することは困難である.

以上のような理由から, 現在の実装では実際に学習イテレーションを数回実行することで, 探索に必要な情報を集めている. また, この profiling 時にはすべての特徴マップに “swap” を割り当てて計算を行っている.

4.3 swap-in の実行タイミングの最適化

提案手法では各データへの属性割り当てに加えて, 通信オーバーヘッド削減のために swap-in の scheduling についても考える. これは図 9 のように各データの swap-in の実行タイミングを早めることができれば, オーバーヘッドを削減できるためである. 図 9 の例では層 4 に対応する swap-in の実行タイミングを早めることで同期によるオーバーヘッドが削減されている.

ここで, backward における swap-in には以下の 2 つの特徴がある.

- 各データの swap-in は GPU メモリにそのデータが収まるだけの余裕がある時点で開始することができる
- 各 swap-in によるオーバーヘッドはその実行タイミングが早いほど発生しにくい

提案手法ではこの 2 点を考慮して, 各 swap-in の開始は GPU メモリに余裕がある限り単純に早めるものとする. backward の処理の各時点での空きメモリ量については, 4.2 節の profiling で取得した情報から判断することができる.

以下で説明する探索において, 各特徴マップへの属性の割り当て方の評価の際には, この swap-in の実行タイミングの最適化を行った上での実行時間・メモリ使用量を用いることとする.

4.4 属性割り当て最適化のための探索

ここからは各データに対する属性 (“keep”/“swap”/“recompute”) 割り当て最適化のための探索について述べる.

4.4.1 探索全体の方針

この探索では NN の計算の実行時間がより短くなるような属性の割り当て方を決定する. しかし, 属性の割り当て方を全探索する場合, 各特徴マップに 3 種類の属性のいずれかを割り当てるために, 層数 n に対して探索空間は $O(3^n)$ と非常に大きい. そこで, 以下で説明するようなヒューリスティクスを用いることで探索空間を削減する.

data-swapping 手法では swap を行ったとしても, 計算時間により通信を隠蔽できるためにオーバーヘッドを削減しやすい. 一方, recompute 手法では再計算によって単純に計算時間が増加する. これらを考慮して, 提案ヒューリスティクスでは次のように探索を 2 段階に分けて行う.

(1) recompute は考えずに keep 対象と swap 対象の選択のみでオーバーヘッドの削減を目指す (4.4.2 節)
(2) swap によるオーバーヘッドと recompute によるオーバーヘッドを比較することで recompute 対象を選択していく (4.4.3 節)

4.4.2 keep 対象と swap 対象の選択

1 段階目の探索としては各特徴マップに対して, “keep” と “swap” のどちらを割り当てるかを決定する. しかし, この探索を全探索で行うと探索空間が大きくなってしまうため, 以下では探索対象の削減を考える.

データを swap する場合, GPU メモリ使用量を削減する代わりに通信オーバーヘッドが発生する可能性がある. つまり, この swap 対象の最適化問題では「実行時間」と「GPU メモリ使用量」の間にトレードオフがある. そこで, swap 対象の選択ではメモリ使用量を考慮して, 基本的には各特徴マップに “swap” を割り当てる. そしてその状態から, swap することによりオーバーヘッドを発生させる特徴マップだけを “keep” に切り替えていくことを考える. よって, swap 対象の選択では以下のような流れで探索を行う.

(1) すべての層の特徴マップに “swap” を割り当てた状態を初期状態とする.
(2) 各特徴マップについて, それぞれ「swap することによりオーバーヘッドを発生させるか」を評価
(3) (2) でオーバーヘッドを発生させないと評価された特徴マップについては “swap” で決定
(4) (2) でオーバーヘッドを発生させると評価された特徴マップに対してのみ “keep” と “swap” のどちらを割り当てるか全探索
(5) 最も実行時間が短くなる属性の割り当て方で決定

この探索の詳細について述べる. まず, (2) の各特徴マップの評価方法について述べる. ここでの評価基準は実行タイムラインをシミュレートした際に「その特徴マップについての swap が完全に計算で隠蔽されているかどうか」である. そして, 通信を隠蔽できないと評価された特徴マップに対してのみ, “keep” と “swap” のどちらを割り当てるかの全探索を行う. また, それ以外の特徴マップについては “swap” で決定する.

図 10 を例にして説明する. 図 10 では層 5・6・7 の特徴マップの swap-out および層 4・6・7 の swap-in(赤枠部分) が計算によって隠蔽しきれていない. よって, この例では

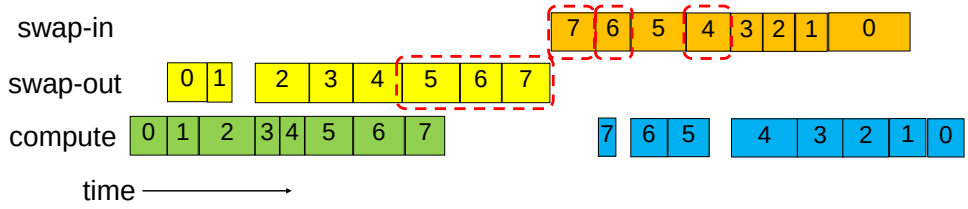

図 10: swap することによりオーバーヘッドを発生させる特徴マップの例．この例では赤枠部分の swap が計算で隠蔽しきれていない．

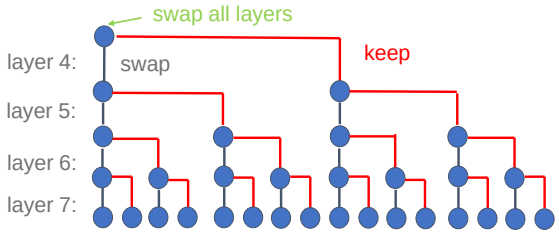

図 11: swap 対象の選択における探索木

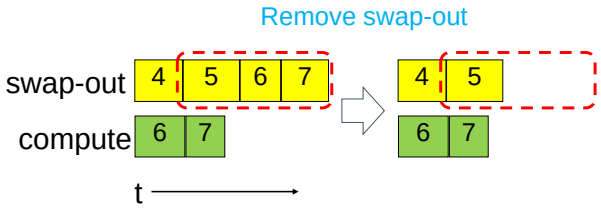

図 12: swap-out の削減．出力層から順 (層 7→6→5 の順) に swap-out を削減する

層 4・5・6・7 の特徴マップが swap によるオーバヘッドを発生させるという評価になる．このとき (4) における探索木は図 11 のようになる．

swap-out を考慮した探索空間の削減

しかし, 実際の NN に対して上記の探索を行ったところ非常に時間がかかってしまった．これは探索対象の削減が不十分だったためである．

この問題を解決するために, さらなる探索空間の削減を考える．そのために我々は実際に data-swapping 手法を用いた際の NN の計算処理の実行タイムラインを調査した．その結果, forward 計算で隠蔽できない swap-out は図 12 左の赤枠部分のように, 実行タイムライン上では forward 処理の最後にすべて連続するということが判明した．これは forward 処理における swap-out は次のような特徴を持つためである．

- 各層の forward 計算は swap-out を待たない
- 対応する forward 計算が終了しないと, 各 swap-out は開始できない

この 2 点より, 隠蔽できない swap-out は実行タイムライン上ではすべて連続するのである．上記の探索ではこれらの swap-out に対応する特徴マップについても全探索を行っていたが, 探索空間削減のためにこれらの特徴マップについては貪欲法を適用することとする．具体的には図 12 右のように単純に後ろから swap-out を削減していく, つまり出力層から順に GPU メモリに収まる限り “keep” を割り当てていく．

また, この貪欲法の適用のために (4) の探索方法を次のように修正する．

- まず, (2) において swap-in でオーバーヘッドを発生させると評価された特徴マップに対してのみ “keep” と “swap” のどちらを割り当てるか全探索する

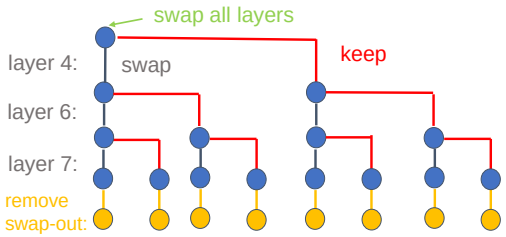

図 13: swap 対象の選択における探索木 (修正後)

- その際の探索木の各葉において, swap-out を隠蔽しきれない特徴マップが残っていれば, 出力層から順に “keep” に切り替えていく．

修正後の探索を図 10 を例にして説明すると, まず swap-in を隠蔽しきれない層 4・6・7 の特徴マップに対しては全探索を行う．そして, その際の探索木の各葉において層 5 などの swap-out を出力層から順に削っていくのである．このとき (4) における探索木は図 13 のようになる．

4.4.3 recompute 対象の選択

2 段階目の探索では, 1 段階目の探索で “swap” を割り当てられた各特徴マップに対して “swap” と “recompute” のどちらを割り当てるかを再決定する (1 段階目の探索で “keep” とされた特徴マップはそのままとする)．

この探索では各特徴マップについて, その特徴マップを「swap することによるオーバーヘッド」と「recompute することによるオーバーヘッド」をそれぞれ比較する．そして, recompute したほうがオーバーヘッドが小さくなる特徴マップについては “recompute” を割り当てていくことを考える．

各特徴マップ X に関する性能オーバーヘッドの比較は以下の関数 $r_{(s,r)}(X)$ を用いて行う．ここで $swap\_overhead(X)$ は X 以外の特徴マップの属性を固定し

表 1: 実験環境

GPU	Tesla V100 (Volta 世代)
GPU メモリ容量	16 GB
CPU	Intel Xeon Gold 6140 (Skylake 世代)
CPU メモリ容量	192 GB
PCI-Express	gen3 x16
OS	CentOS Linux 7.4
CUDA	CUDA 9.1
cuDNN	cuDNN 7.1

た場合に, X を swap することで発生するオーバーヘッドである. 同様に $recompute\_overhead(X)$ は X を recompute することで発生するオーバーヘッドである.

$$r_{(s,r)}(X) = \frac{recompute\_overhead(X)}{swap\_overhead(X)} \quad (1)$$

$r_{(s,r)}(X) < 1.0$ の場合, X は swap するより recompute したほうがオーバーヘッドが小さいということになる. また, $r_{(s,r)}(X)$ が小さいほど “swap” から “recompute” に切り替えた際にオーバーヘッドが削減されやすいと考えられる. 一方で $r_{(s,r)}(X) > 1.0$ の場合は swap したほうがよい. 以上を考慮して, この探索は次のような流れで行う.

(1) 1 段階目の探索で “swap” を割り当てられた特徴マップの集合を $L_{(s,r)}$ とする
(2) $L_{(s,r)}$ が空ならば探索終了
(3) $L_{(s,r)}$ 中の各特徴マップ X についてそれぞれ $r_{(s,r)}(X)$ を計算
(4) $r_{(s,r)}(X) \geq 1.0$ となる X を $L_{(s,r)}$ から取り除く. それらの特徴マップについては “swap” で決定
(5) $r_{(s,r)}(X) < 1.0$ となる X の中で $r_{(s,r)}(X)$ が最小となる X を一つ選び $L_{(s,r)}$ から取り除く. その特徴マップについては “recompute” で決定
(6) (2) へ

5. 実験と評価

我々は深層学習フレームワーク Chainer(v3)[10] を拡張して提案手法を実装した. Chainer は PFN が開発したフレームワークであり, 内部で独自 CUDA カーネルや NVIDIA の cuDNN ライブラリ [12] を呼び出すことで GPU による計算を可能としている. そして, その拡張した Chainer を用いて実験を行い，性能を評価した.

実験環境を表 1 に示す. この環境では GPU メモリ容量は 16 GB である.

また, 以下の実験では ResNet50[5], GoogleNet[13], AlexNet[14] を用いた. この 3 つは全て画像認識用の NN である. ResNet50 と GoogleNet は層数は多いが各層の計算量は小さい傾向がある. 一方, AlexNet は層数は少ないが計算量が大きい層の割合が多い. そして, 各実験において計算の性能は 1 学習イテレーションあたりのバッチサイズ/実行時間 $[\#images/s]$ で評価した.

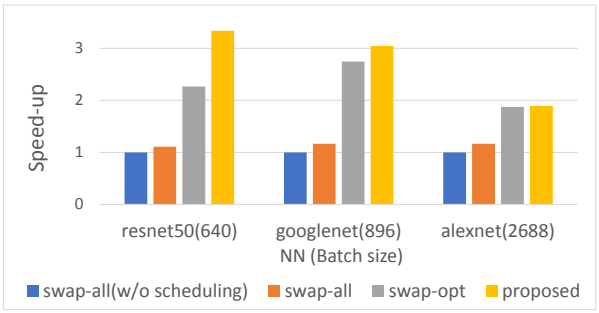

図 14: 各最適化による性能への影響 (NN 毎にそれぞれ swap-all(w/o scheduling) の性能を 1 とした場合の性能を示す)

5.1 各最適化の評価

4 章で述べたように, 我々の提案手法では swap 対象・recompute 対象・各 swap-in の実行タイミングを最適化している. これらの各最適化による性能への影響について評価する. そのため, この実験では以下の 4 つの手法で計算を行った.

- swap-all(w/o scheduling) : 全特徴マップを swap 対象とする. かつ, 各 swap-in は一つ前の計算と同時に開始する
- swap-all : 全特徴マップを swap 対象とする
- swap-opt : 4.4.2 項に基づいて swap 対象を最適化
- proposed : 4.4 節に基づいて swap 対象と recompute 対象を最適化

また, swap-all(w/o scheduling) 以外の 3 手法では 4.3 節に基づいて各 swap-in の実行タイミングの最適化を行った.

この実験では ResNet50, GoogLeNet, AlexNet の計算をそれぞれ GPU メモリ容量を超えるような問題サイズで行った. 結果を図 14 に示す.

図 14 では swap-all は swap-all(w/o scheduling) と比較して 11 – 16% 性能が向上した. これは各 swap-in の実行タイミングが早められたことにより, CPU-GPU 間通信が計算によって隠蔽されやすくなったためである. また, swap-opt では swap 対象が最適化され, 通信されるデータ量が削減されたために swap-all と比較して性能が 1.8 – 2.7 倍に向上した.

そして, どの NN でも data-swapping 手法だけでなく recompute 手法を取り入れた proposed が最も高い性能となった. 特に ResNet50 では proposed の性能が swap-opt の 1.47 倍となった. ResNet50 は特徴マップのサイズに対して計算量が小さい層が多いが, そのような層の特徴マップは swap するより再計算したほうが性能へのオーバーヘッドが小さくなりやすいためである. 一方で AlexNet のような計算量の大きい NN では, swap を十分隠蔽できる計算時間があるために recompute が行われることは少なかった. また, AlexNet については proposed の性能は swap-opt の 1.01 倍であり, 性能向上は小さかった.

5.2 NN アプリケーションへの適用

この実験では ResNet50, GoogLeNet, AlexNet の計算をそれぞれバッチサイズを変化させて行った．また，この実験では各 NN の計算を以下の 3 つの手法でそれぞれ行った．

- in-core：data-swappping と recompute を行わない
- proposed：提案手法により swap 対象・recompute 対象・各 swap-in の実行タイミングを動的に決定
- superneurons：Wang らの提案した SuperNeurons[11] に基づいて，以下のように swap 対象・recompute 対象・各 swap-in の実行タイミングを静的に決定
  - 出力層から順に優先的に特徴マップを GPU メモリ上で保持
  - GPU メモリに収まらない特徴マップのうち，計算量の大きい畳み込み層の特徴マップを swap 対象とする．また，その他の計算量の小さい層の特徴マップを recompute 対象とする
  - 各 swap-in は直前の畳み込み層の計算と同時に開始

ResNet50 での結果を図 15 に示す． この実験ではバッチサイズを 256 以上とした場合，計算に必要なメモリ量が GPU メモリ容量を超えてしまうため in-core では計算を行うことができなかった． また，バッチサイズ $n = 640$ とした場合，計算全体では 50 GB 以上のメモリを必要とした．

図 15 では in-core の性能は最大で 316 [#images/s] であった．これに対して proposed の性能は $207 - 316$ [#images/s] となった．特に GPU 容量を越える問題サイズに対して提案手法を用いた場合は，in-core と比較して $16 - 34\%$ の性能低下で計算を行うことができた．

そして，GPU 容量を越える問題サイズにおいて，proposed は superneurons と比較して $1.33 - 1.84$ 倍の性能となった．提案手法では層の種類ではなく，実際の各層の計算時間および計算全体の処理時間を考慮して swap 対象と recompute 対象を決定している．そのため，superneurons のように静的に決定した場合より性能が向上したのである．

また，図 15 では proposed と superneurons は共にバッチサイズが大きくなるほど性能が低下している．これはバッチサイズに比例してメモリ使用量が増加するために，GPU メモリ上に保持できない特徴マップが増加したことが原因である．つまり，問題サイズが大きいほど，swap 対象または recompute 対象が増えて性能低下につながるのである．

次に GoogLeNet での結果を図 16 に示す． 図 16 では ResNet50 での結果と同様の傾向が見られた．この実験では GPU 容量を越える問題サイズにおいて，proposed は in-core と比較して $2 - 39\%$ の性能低下で計算を行うことができた．また，proposed は superneurons と比較して最大で 1.33 倍の性能となった．

最後に AlexNet での結果を図 17 に示す． 図 17 では in-core と比較して，proposed の性能低下は 5% 以下であった．これは AlexNet は計算量が大きいために，提案手法を用いた際に recompute が行われることが少なく，かつ swap 通信を十分隠蔽できたためである．また，他 2 つの NN での結果と異なり，AlexNet に対しては superneurons でも性能低下は小さく proposed との性能差も小さかった．

以上のように提案手法では，NN の構造や問題サイズに合わせて最適化を行うことにより，静的な手法より性能が向上することが確認できた．

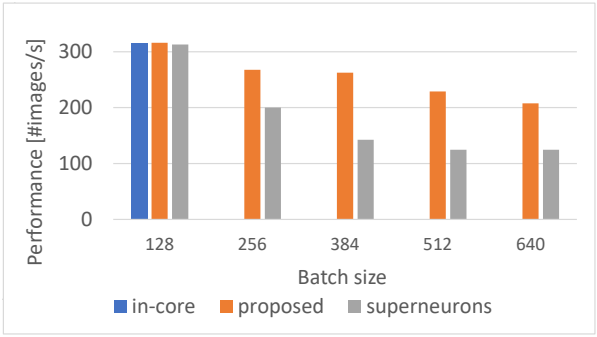

図 15: ResNet50 への適用時の性能

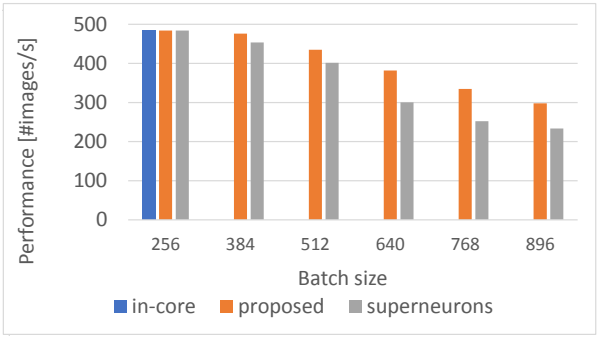

図 16: GoogLeNet への適用時の性能

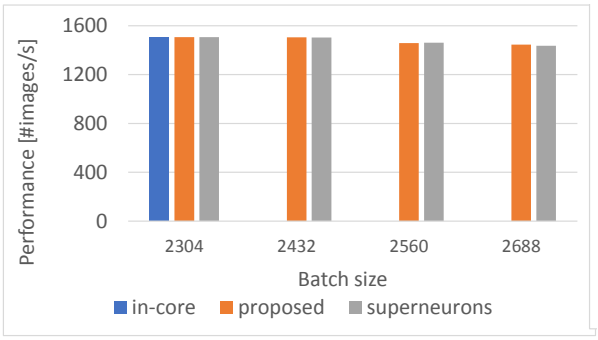

図 17: AlexNet への適用時の性能

6. 関連研究

3.3 節で述べたように Wang らは data-swapping 手法と recompute 手法の両方を用いることを提案している [11]．しかし，swap 対象と recompute 対象を静的に決定している点で本研究とは異なる．

Rhu らは data-swapping 手法を用いて大規模な NN の計算する vDNN を提案している [6]．vDNN では swap 対象を

動的に決定しているが, recompute 手法を取り入れていない. また, Cho らは data-swapping 手法を Caffe や Chainer 上で実装し性能評価を行っている [8]. これらに加えて, Ito らの ooc_cuDNN ライブラリでは各計算およびデータを分割した上で data-swapping を行うことにより, 1 層の計算だけで GPU メモリ容量が足りなくなる場合でも GPU での計算を可能としている [15].

一方, Chen らは recompute 手法を用いることで大規模な NN の計算を行うことを提案しているが, data-swapping は行っていない [9].

7. まとめと今後の課題

本論文では data-swapping 手法と recompute 手法を用いる際の性能オーバーヘッドを削減するための最適化手法について述べた. 提案手法では実行時 profiling に基づいて swap 対象と recompute 対象を最適化することで, CPU-GPU 間通信や再計算によるオーバーヘッドを削減した. また, swap の scheduling による高速化も行った. 提案手法により, GPU 容量を超える NN でも $2-39\%$ の性能低下で計算できることを示した.

提案手法では各学習イテレーションにおいて同じ問題サイズおよび同じ計算を行う NN による深層学習のみを対象としている. そのため, 今後の課題としては RNN のようなイテレーション毎に問題サイズが変化する NN への対応を考えている. また, 動的計画法を用いるなど探索手法の改善をしていきたい.

謝辞 本研究は JST-CREST の研究課題「ポストスケール時代のメモリ階層の深化に対応するソフトウェア技術」の一部支援を受けております.

参考文献

[1] Le, Q. V., Ngiam, J., Coates, A., Lahiri, A., Prochnow, B. and Ng, A. Y.: On optimization methods for deep learning, *International Conference on Machine Learning (ICML)*, pp. 265–272 (2011).
[2] Radford, A., Metz, L. and Chintala, S.: Unsupervised representation learning with deep convolutional generative adversarial networks, *International Conference on Learning Representations (ICLR)* (2016).
[3] Frank Seide, Gang Li, D. Y.: Conversational Speech Transcription Using Context-Dependent Deep Neural Networks, *Annual Conference of the International Speech Communication Association (Interspeech)*, pp. 437–440 (2011).
[4] Vinyals, O. and Le, Q. V.: A neural conversational model, *Deep Learning Workshop in conjuction with ICML* (2015).
[5] Simonyan, K. and Zisserman, A.: Very deep convolutional networks for large-scale image recognition, *International Conference on Learning Representations (ICLR)* (2015).
[6] Rhu, M., Gimelshein, N., Clemons, J., Zulfiqar, A. and Keckler, S. W.: vDNN: Virtualized Deep Neural Networks for Scalable, Memory-Efficient Neural Network Design, *IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pp. 1–13 (2016).
[7] Meng, C., Sun, M., Yang, J., Qiu, M. and Gu, Y.: Training Deeper Models by GPU Memory Optimization on TensorFlow, *ML Systems Workshop in NIPS* (2017).
[8] Cho, M., Le, T. D., Finkler, U. A., Imai, H., Negishi, Y., Sekiyama, T., Vinod, S., Zolotov, V., Kawachiya, K., Kung, D. S. and Hunter, H. C.: Large Model Support for Deep Learning in Caffe and Chainer, *SysML Conference* (2018).
[9] Chen, T., Xu, B., Zhang, C. and Guestrin, C.: Training Deep Nets with Sublinear Memory Cost, `https://arxiv.org/abs/1604.06174` (2016).
[10] Tokui, S., Oono, K., Hido, S. and Clayton, J.: Chainer: a Next-Generation Open Source Framework for Deep Learning, *Machine Learning Systems Workshop in conjuction with NIPS* (2015).
[11] 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, *Principles and Practice of Parallel Programming (PPoPP)* (2018).
[12] Chetlur, S., Woolley, C., Vandermersch, P., Cohen, J., Tran, J., Catanzaro, B. and Shelhamer, E.: cuDNN: Efficient Primitives for Deep Learning, `https://arxiv.org/abs/1410.0759` (2014).
[13] Szegedy, C., Liu, W., Jia, Y., Sermanet, P., Reed, S., Anguelov, D., Erhan, D., Vanhoucke, V. and Rabinovich, A.: Going Deeper With Convolutions, *The IEEE Conference on Computer Vision and Pattern Recognition (CVPR)* (2015).
[14] Sutskever, A. K. I. and Hinton, G.: ImageNet Classification with Deep Convolutional Neural Networks, Advances in neural information processing systems (NIPS)，pp. 1097–1105 (2012).
[15] Ito, Y., Matsumiya, R. and Endo, T.: ooc_cuDNN: Accommodating Convolutional Neural Networks over GPU Memory Capacity, *IEEE International Conference on Big Data* (2017).