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

根岸 康　今井 晴基　土井 淳　河内谷 清久仁

ディープラーニングのネットワークモデルは年々大きくなる一方、GPU 上のメモリ容量は計算速度程向上していない。一方、高解像度イメージの処理や、3D イメージの処理等ディープラーニングの入力データの大規模化への需要は高まっており、近い将来ミニバッチサイズを 1 としても GPU 上のメモリに全てのデータが乗らないことも想定される。

処理に必要なメモリ量が GPU メモリの量を超える場合、一般に CPU-GPU 間で必要なデータを明示的に転送する複雑なプログラムが必要となるが、Unified Memory の導入により、最小限の変更で GPU メモリを意識しないプログラミングが可能になる。

しかし、Unified Memory をこのような目的で使用した場合の性能測定結果はあまりない。このため、(1) Unified Memory 導入によりどの程度まで大きなメモリを使用するプログラムを実行可能か (2) Unified Memory 導入に伴う実行速度の低下が実用的な範囲であるか (3) ヒントにより十分な性能向上が得られるのかについての検証が必要となる。

我々は、Chainer を用いて Unified Memory 使用時の実行時間を検証し、(1) Unified Memory の使用により、同一モデルで 4-28 倍のミニバッチサイズ、同一ミニバッチで 4-81 倍の入力サイズに対応するモデルを実行可能であること (2) 使用するメモリ量が実メモリより十分小さい場合、Unified Memory による性能低下はほとんどないこと (3) 使用するメモリ量が実メモリより大きい場合、性能低下は 5-10 倍程度であること (4) 現状の CUDA8 の Unified Memory の実装及び API ではヒントにより Unified Memory の性能を向上させることは困難であることを確認した。

以下、第 1 章で背景について、第 2 章で Unified Memory を使用したソフトウェア開発について、第 3 章で性能測定及びその結果について、第 4 章で最適化の方針について、第 5 章でまとめについて述べる。

Deep learning models are becoming large, but GPU memory size is not becoming large. Accordingly the learning model will not fit into the GPU memory in the near future.

When the model does not fit in the GPU memory, complex programing is required to transfer the data between CPU and GPU explicitly. We resolve this issue by introducing Unified Memory, but there are little reports about the performance with Unified Memory.

This paper discuss the performance with Unified Memory. In this paper, we found (1) Unified Memory supports 4-81 times larger memory than normal execution, (2) when the memory usage is enough less than the GPU memory, the peformance depreciation is small (3) when the memory usage is larger than the GPU memory, the performance becomes worse by 5-10 times, and (4) it is difficult to improve the performance by inserting hint, according to the current implementation of CUDA8.

1 背景

ディープラーニングでは、精度の向上のためより多くのが用いられるとともに、入力データ、モデルの大規模化が進んでいる。ディープラーニングでは GPU が用いられれることが多いが、GPU 上のメモリの大きさは計算能力の向上と比較するとあまり拡大していない。

ディープラーニングのモデルは大規模化が進んでいる。例えば、Resnet [3] では 152 層が使用されており近年他のネットワークでもこれまでよりより多くの層が使われている。同様に入力データについても、医

Consideration on Performance of Large model Deep Learning Model wih Unified Memory and its API

Yasushi Negishi, Haruki Imai, Jun Doi, and Kiyokuni Kawachiya, 日本アイ・ビー・エム (株) 東京基礎研究所, IBM Research - Tokyo.

療等でより解像度が高い画像データを直接入力として使用する要求が高まっており、各層が必要とするメモリ量は拡大している。一方、GPU のメモリの量はあまり拡大していない。Nvidia 社の場合でも次世代 GPU VOLTA となっても搭載メモリの量は当面現行と変わらず 16GB の予定である。GPU 用のメモリには HBM(High Bandwidth Memory) と呼ばれる高速だが高価なメモリが使用されるため、この傾向は当面維持されると考えられる。

この様な状況下では、ミニバッチサイズを 1 としても使用メモリが GPU の物理メモリの量を超えることが現実の問題となりつつある。例えば、入力データとして 4K(スーパーハイビジョン) 画像 (3840x2100) を近年良く利用される ResNet152 というモデルを修正して適用した場合を考えると、必要となる GPU メモリは 21-40 GB となり [†1]。次世代の GPU メモリ VOLTA の量を上回る。

我々の目標は、このような GPU メモリの物理的上限を意識することなく Deap Learning Framework を実装・利用できる環境を用意することにある。

2 Unified Memory を使用したソフトウェア開発

2.0.1 Unified Memory の概要

Unified Memor は、Nvidia 社の提供する GPU を使用するためプラットフォーム CUDA [7] において、主にホストメモリと GPU メモリを統一的に取り扱うために提供されていおり、メモリ確保用に特別な API CudaMemAllocManaged を使用することで、ホスト側からのメモリアクセスと GPU 側からのメモリアクセスを同一のポインタで表現することが可能になる。

この機能は、本来、同一データをホストと GPU の両方からアクセスする場合を想定して、同一データに対する CPU メモリ用 GPU メモリ用の複数の allocation や明示的な CPU-GPU 間のメモリ転送に関する記述を省き、より簡潔に記述するための機能である。

†1 入力データサイズは元の画像サイズ 256x256 の 123.4 倍で、重み数 6GB・データ数 15G となる。16bit 浮動小数点 (2 バイト) で 1-2 セット保持と仮定

本論文でも Unified Memory を使用するが、その使用方法は上記のものとは異なり、CPU からのメモリアクセスは想定せず GPU メモリを実メモリサイズ以上に使用 (オーバーコミット) するために使用する。Unified Memory の機能は CUDA6.0 から導入されていたが、現行の PASCAL 世代 GPU からは、GPU page fault がサポートされ、On-demand で CPU-GPU 間のページ移動が行えるようになり、実行時に GPU のメモリが不足する場合に、CUDA のランタイムシステムが自動的に CPU-GPU 間のメモリ移動を行い、GPU プログラムからは仮想的に実メモリ以上の大きなメモリの使用が可能になった。

GPU 上にメモリ確保する API である CudaMemAllloc の全ての呼び出しを CudaMemAllocManged に変更することで、通常のメモリに変わって Unified Memory を使用できる。

更に次の VOLTA 世代の GPU では、Unified Memory の Cache coherency が向上し、atomic 操作も行えるようになり、Unified Memory はより一層活用されるようになると考えられる。

2.0.2 Unified Memory を使用したソフトウェア開発

処理に必要なメモリ量が GPU メモリの量を超える場合、Unified Memory の使用によりソフトウェア開発が容易になる。

Unified Memory を使用しない場合
CPU-GPU 間で必要なデータを逐次明示的に転送するプログラムを作成する必要がある。

1. **コーディング** GPU の実メモリの量を考慮し、転送対象とするデータをプログラミング前に確定する必要がある。
2. **機能デバッグ** データ転送を含むプログラミングが終了するまで機能デバッグを開始できない。
3. **性能最適化** 方針・プログラムが妥当であれば、機能デバッグ終了時点である程度の性能は得られる。

Unified Memory を使用する場合
より単純なプログラミングが可能になる。

1. **コーディング** GPU の実メモリの量を考慮せずにプログラムを作成する。

2. **機能デバッグ** Unified Memory を使用して動作させて機能をデバッグする。
3. **性能最適化** 実行時間・メモリ使用量を測定し、メモリ転送が必要な箇所や性能ボトルネック部分にヒントを与えることで性能最適化する。

3 測定方針

このように Unified Memory の利用によりソフトウェア開発は容易になるが、実際のプログラムが、GPU の実メモリ量をどの程度まで超えても動作するのか、またその時の実行速度について報告した研究はこれまであまり無かった。

報告が少ない理由の一つは、現行の GPU の物理メモリの範囲内で効率よく実行されるモデルのみがよく使用されることにある。

本論文では、将来の利用状況を想定して、次の 2 つの方法で GPU メモリが大きく消費される状況を設定し、プログラムを実行して実行可能性や速度について議論する。

1. **ミニバッチサイズ変更** ミニバッチサイズの増加によりメモリ使用量を増加させる。モデルは修正しない。入力サイズは元のサイズ (224) に固定する。
2. **入力サイズ変更** モデルの一部を修正し入力データのサイズを変化させる。ミニバッチサイズは固定 (今回は 32) する。

次節でモデルの修正方法について説明する。

3.1 ディープラーニングモデルの修正

3.1.1 モデル修正の方針

本節では、大きなサイズの入力データに対応するためのモデルの修正方法について説明する。今回は、層の数は変更せずに各層の大きさのみを修正する。本論文では元のモデルに対して最小限の修正で使用メモリを任意の大きさに増加させることを目的とし、精度の向上は考えない。

3.1.2 モデル修正方法

GoogleNet、ResNet 等多くの Convolution Network が、前半が Convolution 層、後半が全結合層から構成されている。この内、Convolution 層だけを入力サイズに合わせて修正し、全結合層のサイズは変更しない。

Convolution 層は入力サイズに比例してメモリサイズと計算量が増加する一方、全結合層は、サイズの二乗に比例して、メモリサイズと計算量が増加するので、入力サイズに合わせて全結合層のサイズを拡大することは現実的ではない。

Convolution と全結合層のリンクの数を合わせるために、Convolution 層と全結合層を結合する層 (例えば、max pooling 層) のパラメータを調整する。

以下、比較的単純な alexnet の修正を例にとって説明する。この例では初期化の際に 224 の定数倍の値を insize として与えている。Chainer では入力行列のデータサイズに合わせて各層の定義が決定されるので、コード上での明示的な変更が必要な部分は、以下の例では max pooling 層のパラメータのみである。

```
import chainer
import chainer.functions as F
import chainer.links as L


class Alex(chainer.Chain):
    insize = 224

    def __init__(self, insize=224):
        Alex.insize=insize
        super(Alex, self).__init__(
            conv1=L.Convolution2D(3,  64, 11, stride=4, pad=2),
            conv2=L.Convolution2D(64, 192,  5, pad=2),
            conv3=L.Convolution2D(192, 384,  3, pad=1),
            conv4=L.Convolution2D(384, 256,  3, pad=1),
            conv5=L.Convolution2D(256, 256,  3, pad=1),
            fc6=L.Linear(256 * 6 * 6, 4096),
            fc7=L.Linear(4096, 4096),
            fc8=L.Linear(4096, 1000),
        )

    def forward(self, x):
        h = F.max_pooling_2d(F.relu(self.conv1(x)), 3, stride=2)
        h = F.max_pooling_2d(F.relu(self.conv2(h)), 3, stride=2)
        h = F.relu(self.conv3(h))
        h = F.relu(self.conv4(h))
        #h = F.max_pooling_2d(F.relu(self.conv5(h)), 3, stride=2)          #orig
        h = F.max_pooling_2d(F.relu(self.conv5(h)), 3 * Alex.insize//224,  #fixed
                             stride=2 * Alex.insize//224, cover_all=False) #fixed
        h = F.relu(self.fc6(h))
        h = F.relu(self.fc7(h))
        return self.fc8(h)
```

今回の測定では、ConvNet ベンチマーク及びその PR に含まれる以下のモデルを使用した。

1. alexnet
2. googlenet
3. resnet50
4. resnet101
5. resnet151
6. vgga
7. vg16
8. vg19

上記 PR 中の resnet50, resnet101, resnet151 につ

いては、上記手法と同様の方法で入力サイズ変更に対応済であったのでそれを使用した。それ以外のモデルに関しては入力サイズ変更のための修正を適用した。

3.2 測定環境

3.2.1 ハードウェア

計測には以下のノードを使用した。

システム名 IBM Power System S822LC for High Performance Computing (Minsky) [2]

CPU IBM POWER8 2 基 (計 20 コア) 搭載、システム全体のメモリ 256GB

GPU NVIDIA Telsa P100 4 基搭載、一基当りのメモリ 16GB

このシステムでは NVLink のインタフェースを持つ POWER8 プロセッサが使用されている。各 GPU-GPU 間が NVLink 2 brick を使用して片方向 40GB/s、両方向 80GB/s で結合されている。

x86 プロセッサの場合、NVLink のインタフェースを持たないため、GPU-CPU 間は PCI Express 経由で GPU-CPU 間は 32GB/s で接続される。

このため、X86 プロセッサを使用した場合、Unified Memory の性能に影響を及ぼすことが想定されるが、本論文では X86 プロセッサを使用した場合の性能は測定できなかったため、今後の課題としたい。

3.2.2 ソフトウェア

今回使用したソフトウェアは以下の通り。

RHEL7.3

CUDA9.0

Python3.0.6

Chainer Chainer [4] は、Preferred Netoworks によって開発されたディープラーニングフレームワークで、Python によって記述されている。本論文では現時点で最新の Version 2 を用いた [5]。Chainer は Cupy [6] と呼ばれる Numpy 互換ライブラリを経由して GPU にアクセスする。Cupy ではメモリオブジェクトが抽象化されているため統一的な書き換えが容易であり、Unified Memory を使用するには Cupy の Memory Class を修正する。Chainer V2 はまだ Unified Memory 対応されていないので、本論文では Chainer V1 向けの修正と同様の修正を Chainer V2 に行った。また、CUDA9 に対応するための修正を行った。今回の測定では Chainer を使用したが、本研究によって得られる知見の多くは他のフレームワークにも適用可能であると考えている。

ConvNet Benchmark 異なるディープラーニングフレームワーク間で性能比較するためのベンチマーク [1]。入力データはベンチマークプログラムが乱数で生成するため、入力データを個別に用意する必要はない。一方、学習精度・検証精度を測定することはできない。ベンチマーク自身に alexnet, googlenet, overfeat(標準的な入力サイズが異なるため本論文では不使用), vgga のモデル定義が含まれている。Chainer V2 に対応するために [8] を適用した。この branch には vgg16 のモデル定義、及び、入力サイズ変更に対応した ResNet50, ResNet101, ResNet152 の定義が含まれている。

3.3 測定結果

3 章の方針に基づき、(1) ミニバッチサイズ変更、(2) 入力サイズ変更の 2 つの方針で実行の成否と性能を測定した。本節ではその測定結果について説明する。

3.3.1 ミニバッチサイズ変更

モデルは修正せず、入力サイズは元のサイズの 224x224 (pixel) * 3 (colors) のまま固定した。ミニバッチサイズを変化させてベンチマークを実行し、実行の成否と実行時間を測定した。

最大ミニバッチサイズ (表 3.3.1)

最大ミニバッチサイズの拡大率は、モデルによって異なり 4-28 倍であることを確認できた。

Unfieid Memory による拡大率は層の数が大きいほど大きくなる傾向がある。これは、層の数が大きいほどメモリ全体に対するワーキングセットの割合が小さくなるため、少ないデータ転送で実行が続けられ実行時のエラーを回避可能であるためと考えられる。

実行速度

ミニバッチサイズを変更させた場合の実行速度を図 2 に示す。

モデル	alexnet	googlenet	ResNet50	ResNet101	ResNet152	vgga	vgg16	vgg19
層の数	8	22	50	101	152	11	16	19
最大ミニバッチ数 (オリジナル)	2272	256	96	64	32	160	96	96
最大ミニバッチ数 (Unified Memory)	11072	1984	1248	1056	896	640	640	640
最大ミニバッチサイズの拡大率	4.873	7.750	13.000	16.500	28.000	4.000	6.667	6.667

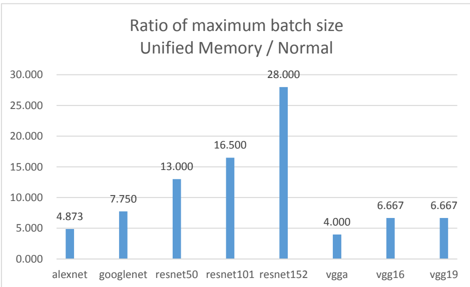

図 1　最大ミニバッチ数の拡大率

この内 GoogLeNet の実行時間を示したものが図 3 である。

また、GoogLeNet のイメージ 1 枚当りの実行時間を図 4 に示す。

注目すべき点は、以下の通り。

1. Unified Memory の使用時、ミニバッチサイズ 224 までは Unified Memory を使用しない場合と実行時間がほぼ同じである。
2. このミニバッチサイズ 224 は Unified Memory を使用しない場合に実行可能なミニバッチサイズ 256 の 87%にあたる。
3. Unified Memory の使用時、ミニバッチサイズ 256 を超えると 1 イメージあたりの実行時間が急激に悪化し、Unified Memory を使用しない場合の実行時間 (約 3msec) の約 10 倍となる。batch を大きくすると更に徐々に悪化する。

Unified Memory を使用した場合、1 イメージ当りの実行時間は大きく低下する。ミニバッチ数の拡大はデータパラレルの導入によっても可能であるため、ミニバッチ数拡大のために Unified Memory の導入することに実用的な意味はない。

3.3.2　入力サイズ変更

モデルの一部を修正し入力データのサイズを変化させた。修正の方法は 3.1.2 節の通り。ミニバッチサイズは十分な実行効率が得られる最小のものと考え、今回は 32 に固定した。

最大入力サイズ

最大入力サイズは、Unified Memory の使用により、4(2x2) 倍から 81 倍 (9x9) に拡大されることが確認できた。

実行速度

入力サイズを変更した場合の実行速度を図 6 に示す。

この内 GoogLeNet の実行時間を示したものが図 7 である。

また、GoogLeNet のイメージ 1 枚当りの実行時間を図 8 に示す。

Unified Memory の導入により実行時間の低下が見られるが、ミニバッチサイズは 1 でも実行不可能な場合にモデルの実行に対しては、Unified Memory の導入は有効な手段の一つとなる。

4　最適化の方針

Unified Memory により使用可能なモデルが大きく拡大可能であることが確認できた。一方、実行速度は大きく低下している。本章では実行速度の最適化の方法を検討する。

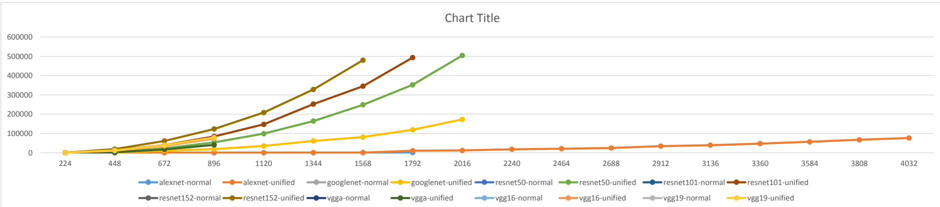

図 2　ミニバッチサイズ変更

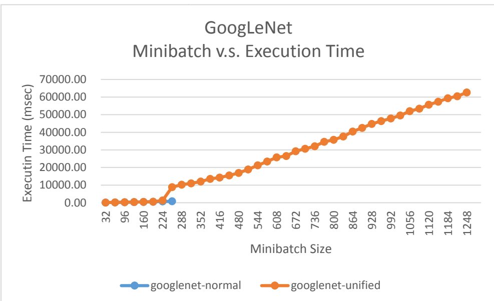

図 3　ミニバッチサイズ変更 (GoogLeNet)

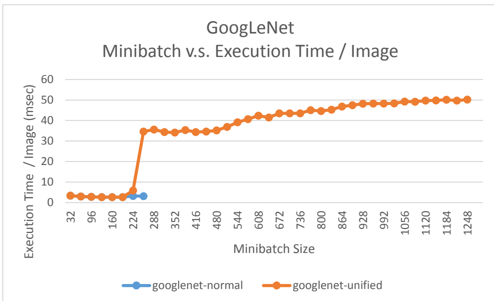

図 4　1 イメージあたりの実行時間 (GoogLeNet)

4.1　ディープラーニングフレームワークのメモリ使用パターン

GoogleNet、ResNet 等の通常の Convolution Network は Forward Propagation と Backward Propagation の 2 つのフェーズによって処理される。メモリ管理の方針はフレームワークによって異なるが、ここでは Chainer を例にとって説明する。

Chainer(Version 1) のメモリ確保・開放部分にト

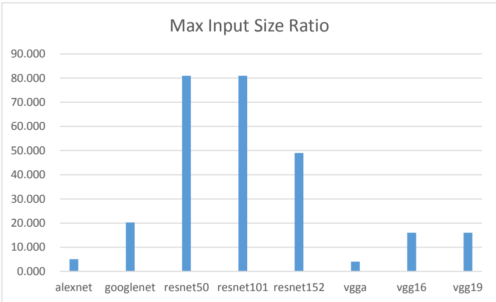

図 5　入力サイズ変更

レースを入れて、時間経過毎のメモリ使用量を計測した。

図 9 に、GoogLeNet(入力サイズ 2240x2240) の 10 iteration 実行時のメモリ確保・開放回数とメモリ使用量のグラフを示す。

グラフの青腺が実際に使用されているメモリの使用量、オレンジ線が Memory Pool を含めた全ての確保されているメモリ量の合計を示している。Memory Pool は未使用のメモリを一旦プールするための領域であり、メモリ確保・開放のコストを軽減するために導入されている。

このグラフから以下の点が読み取れる。

メモリの最大確保量　Unified Memory の導入により、実メモリの 16 GB を超えて約 70GB までメモリを確保できることが確認できる。

イテレーション内のメモリ使用パターン　Chainer では Forward フェーズ中にメモリを確保し、Backward フェーズ中にメモリを開放している。このため、GPU の使用メモリの量は各 iteration の開始時点でほぼ 0、Forward 時から Backward 時の折り返しの時点で最大となる。Backward での参照の終了毎に順次開放され、iteration の終了時点で再びほぼ 0 となる。

Memory Pool によるメモリ使用量　今回の測定では、確保したメモリの約 40%が Memory Pool によって使用されている。Memory Pool 中のメモリはアクセスされることは無いが、CUDA システムによるデータ転送の対象となるので、大きなデータを Memory Pool 用に確保していることは、性能に影響を与える。

これらの観察から得られる最適化の方針は以下の通り。

最適化方針 1　Unified Memory の使用時は Memory Pool は使用しない。

更に、メモリー確保の呼び出し元にもトレースを入れて、どのモジュールがメモリを使用しているか分類した。図 9 に、GoogLeNet(入力サイズ 224x224) の 10 iteration 実行時のメモリ確保・開放回数とモジュール別のメモリ使用量を示す。また、図 9 に、モジュール別の合計使用メモリ量を示す。

これらのグラフから得られる知見は以下の通り。

メモリ使用モジュール　cudnnActivationforward 等少数のモジュールが使用メモリのほとんどを占めており、10 個のモジュールで全体の 90%のメモリ使用を説明できる。また、ログを直接観察すると、モジュールあたり数個程度の非常に大きなメモリを確保していることがわかる。

層に依存しないメモリ使用　アプリケーション等層に依存しないモジュールからのメモリ使用量は少ない。

以上の知見から以下の最適化方針が考えられる。

最適化方針 2　大きなメモリの確保にのみ Unified Memory を使用する。

図 12 にここまでの最適化の方針に従って、最適化を行った結果を示す。MemoryPool の使用を止め、4KB 以下のメモリ確保に通常のメモリを使用するようにプログラムを変更した。これらの最適化により最大ミニバッチサイズは 10 から 9 に減っているが、実行時間は約 15%削減されている。

4.2　ヒントによる最適化

プログラムへのヒントの挿入による最適化について検討する。

4.2.1　必要なヒントの種類

ディープラーニングは Forward フェーズ、Backward フェーズの順で行われ、Backward フェーズでは、対応する Foraward フェーズによる計算結果にア

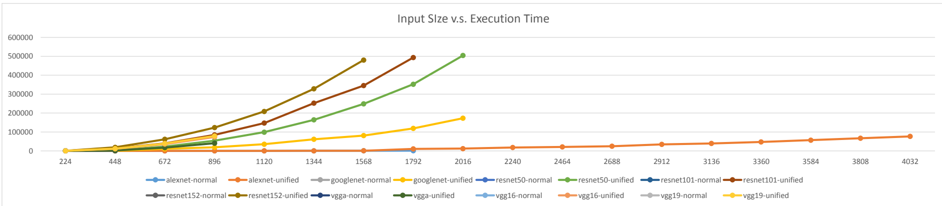

図 6　実行時間 (GoogLeNet)

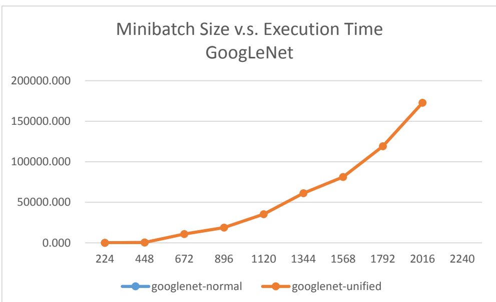

図 7　実行時間 (GoogLeNet)

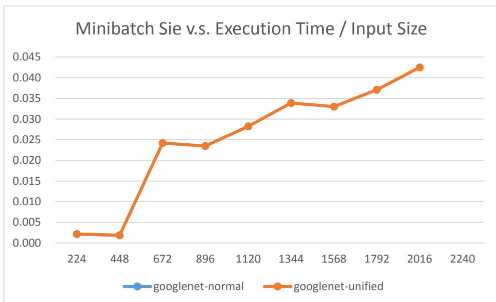

図 8　入力サイズ当りの実行時間 (GoogLeNet)

クセスする。

このアクセスパターンを考えると、以下の 2 種類のヒントによりデータ転送と GPU での計算を並列に行うことが性能改善に有効である。

swap out ヒント　GPU から CPU へのメモリ swap-out のヒントを forward フェーズでの各層の計算終了時点に挿入する。

prefetch ヒント　CPU から GPU へのメモリ

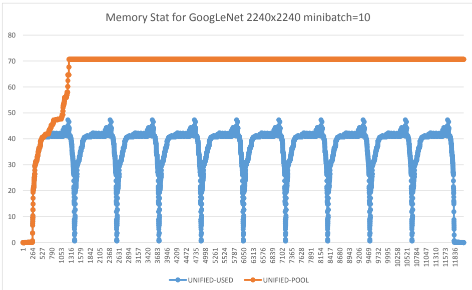

図 9　**GoogleNet(入力サイズ 2240x2240) のメモリ使用パターン**

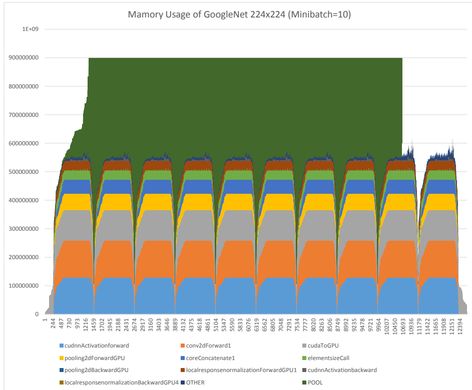

図 10　**GoogleNet(入力サイズ 224x224) のモジュール別メモリ使用パターン**

prefetch のヒントを backward フェーズでの各層の計算開始の適切な時間分前に挿入する。

以下、上記 2 種類のヒントを CUDA の API によって実装可能であるかについて検討する。

4.2.2　ヒントの実装についての検討

CUDA は Unified Memory の使用時にデータ転送を行う際のヒントを与えるために次の API を提供している。このうち、cudaMemPrefetchAsync は上記の prefetch ヒントに相当する。

cudaMemAdviseSetPreferredLocation　メモリを確保するデバイス (ホスト・GPU) の指定

cudaMemAdviceUnsetPreferredLocation　上記ヒントの削除

cudaMemAdviseSetAccessedBy　次にアクセスするデバイス (ホスト・GPU) の指定。

cudaMemPrefetchAsync　非同期にメモリ転送を指示

我々はこれらの API を使用して、上記 2 種類のヒントを実装することを検討した。

このうち、cudaMemAdviseSetPreferredLocation/cudaMemAdviceUnsetPreferredLocation を使用したが、今回の測定条件では使用の有無で性能に有意な差は見られなかった。また、cudaMemAdviseSetAccessedBy については、呼び出し時点でデータ転送が同期的に行われるため, GPU での計算時間を削減する効果は確認できたが、API の CPU-GPU 間のデータ転送が API の呼び出し時点に移動しただけで、全体としての実行時間の削減を実現することはできなかった。

CudaMemPrefetchAsync については、Backward フェーズへの導入により一定のデータ削減効果は確認できた。図 12 に CudaMemPrefetchAsync の効果に

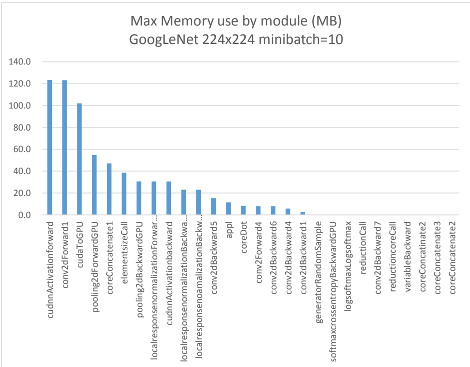

図 11　GoogleNet(入力サイズ 224x224) のモジュール別合計メモリ使用量

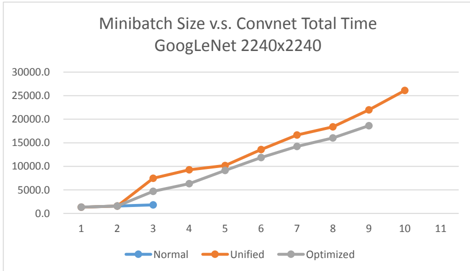

図 12　GoogleNet(2240x2240) における最適化の効果

ついて示す。最大 5%の実行時間削減効果が得られているが、ミニバッチサイズによっては逆に実行時間が増加している。

このように CudaMemPrefetchAsync には一定の効果が見られるが、限度がある。その原因としては、prefetch の代償として swap out するメモリを制御していないことにある。

swap out するメモリを制御するために cudaMemPrefetchAsync を逆方向に使用して、GPU のメモリを非同期にホストに転送することを試みた。

これにより対象メモリのデータがホスト上に転送され、GPU 上のメモリはデータ転送なしにいつでも再使用できる状態となる。このために再利用の際の優先度が上がることが期待される。

予備的な実験を行いこのときの動作を確認した。実験では GPU のメモリ量を超えたメモリを GPU 上で使用し、ディープラーニングを模したアクセスパターンに基づいて、メモリ確保・開放及びメモリアクセスを行った。

予備的な実験の詳細については本論文では触れないが、実験により以下の知見を得た。

1. swapout されるデータはメモリ確保 (allocation) の順序に基づいておこなわれる。アクセスの順序は swapout の順序を決定する際に考慮されない。
2. GPU 上のメモリが不足すると、メモリは最も古く確保されたメモリから順次ホスト上に転送され再利用される。
3. 再利用が一巡するとラウンドロビン順で最も古く確保されたから順次再度ホストに転送し再利用する。
4. メモリ上のデータがホストに転送済みですぐに再利用可能であっても、上記再利用の順序に影響

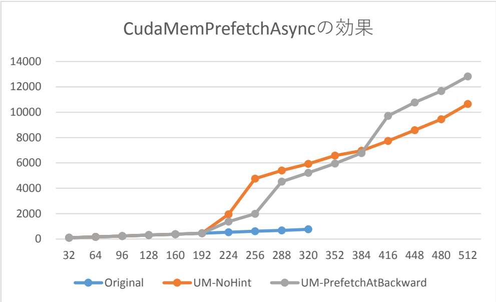

図 13 GoogleNet(2240x2240) における CudaMemPrefetchAsync の効果

を与えない。

5. cudaMemFree によって開放されたメモリであっても、すぐには再利用されず上記再利用の順序で再利用される。

以上の実験を鑑みて、我々は GPU の Unified Memory の再利用の順序を現状の CUDA API を使用して任意に制御することは不可能であると考えている。

次の VOLTA 世代の GPU では、Unified Memory の Cache coherency が向上し、atomic 操作も行えるようになる。これに伴って CUDA の Unified Memory の機能も改善され、メモリ再利用の順序も制御できるようになることを期待している。

5 まとめ

本論文では、、(1) Unified Memory 導入によりどの程度まで大きなメモリを使用するプログラムを実行可能か (2) Unified Memory 導入に伴う実行速度の低下が実用的な範囲であるか (3) ヒントにより十分な性能向上が得られるのかについて、検証した。

我々は、Chainer を用いて Unified Memory 使用時の実行時間を検証し、(1) Unified Memory の使用により、同一モデルで 4-28 倍のミニバッチサイズ、同一ミニバッチで 4-81 倍の入力サイズに対応するモデルを実行可能であること (2) 使用するメモリ量が実メモリより十分小さい場合、Unified Memory による性能低下はほとんどないこと (3) 使用するメモリ量が実メモリより大きい場合、性能低下は 5-10 倍程度であること (4) 現状の CUDA8 の Unified Memory の実装及び API ではヒントにより Unified Memory の性能を向上させることは困難であることを確認した。

今後は、VOLTA GPU 及び CUDA9 を使用してディープラーニングフレームワークで Unified Memory を使用する際の実行性能の最適化、及び、最大入力サイズの拡大を進めて生きたい。

参 考 文 献

[1] Chintala, S.: https://github.com/soumith/convnet-benchmarks.
[2] IBM: IBM Power System S822LC for High Performance Computing, https://www.ibm.com/systems/jp-ja/power/hardware/s822lc-hpc/index.html.
[3] Kaiming He, Xiangyu Zhang, S. R. J. S. M. R.: Deep Residual Learning for Image Recognition, https://arxiv.org/abs/1512.03385.
[4] Networks, P.: Chainer, https://chainer.org/.
[5] Networks, P.: Chainer github, https://github.com/chainer/chainer.
[6] Networks, P.: cupy, https://github.com/cupy/cupy.
[7] NVIDIA: CUDA Toolkit, https://developer.nvidia.com/cuda-toolkit.
[8] Saito, S.: a branch of convnet-benchmarks.