Vol.2018-HPC-164 No.7
2018/5/7

CUDA Unified Memory のディープラーニングへの適用についての考察

根岸康[†1]　今井晴基[†1]　土井淳[†1]　河内谷清久仁[†1]

概要：ディープラーニング分野では高解像度イメージや 3D イメージの処理等大規模モデルへの需要が高まっており，使用 GPU メモリ量は年々拡大している．CPU プログラミングでは仮想記憶により開発者や利用者が実メモリ量を意識することはないが，GPU では開発者や利用者が実メモリの大きさに合わせてパラメータを調整することが必要となる．これは計算効率を重視する GPU プログラミングの特性上ある程度止むを得ないが GPU プログラミングの生産性を向上する上の障害の一つとなっている．NVIDIA 社による GPU プログラミング環境である CUDA では CPU と GPU メモリを仮想的に同一アドレスとして扱う Unified Memory と呼ばれる機能が提供されており，これにより CPU-GPU 間での明示的なデータ転送を隠蔽したプログラミングが可能になっている．Unified Memory は主に HPC 分野での同一データへの CPU と GPU の協調計算用にデザインされており OpenMP 等を介して効果的に用いられている．一方ディープラーニング分野ではほぼ全ての計算が GPU 上で行われるが，Unified Memory はこの目的には設計されておらずあまり活用されていない．Unified Memory 機能ではメモリの確保手順の変更のみで GPU メモリを仮想的に拡張でき，フレームワークへの局所的な変更で大規模モデルへの対応が可能になるが，その際実行速度が問題となる．
Unified Memory を用いた場合の実行速度は CUDA ランタイムの振る舞いに大きく影響を受けるがその詳細は公開されていない．
本稿では，Chainer で使用する数値演算ライブラリ Cupy に Unified Memory を制御する API を追加し，Unified Memory の振る舞いを調査するテストプログラムを作成し，Power9 及び VOLTA GPU 上で動作を検証した．その調査結果に基づいて，Unified Memory を用いてディープラーニングフレームワークを効率的に実装する方法について考察する．

キーワード：深層学習，大規模ニューラルネットワーク，Out-of-core 学習，GPU，Chainer

Consideration on Application of CUDA Unified Memory to Deep Learning

YASUSHI NEGISHI[†1]　HARUKI IMAI[†1]
JUN DOI[†1]　KIYOKUNI KAWACHIYA[†1]

Abstract: Deep Learning model size is becoming larger to process high resolution images or 3D images. On CPU, developers and users do not consider the physical memory size thanks to virtual memory mechanism. As for GPU programming, they should adjust parameters to fit the memory to the physical memory size. It is necessary for calculation efficiency to consider the physical memory size, but is an obstacle to improve the productivity of GPU programming. NVIDIA provides a mechanism, called Unified Memory, to manage CPU and GPU memory in an integrated way by using virtual memory mechanism, which enables programming without explicit data transfer between CPU and GPU. Unified Memory is designed for heterogenous processing with CPU and GPU and is mainly used for HPC area through OpenMP. On the other hand, for Deep Learning area, almost all processing is done by GPU only, and Unified Memory is not designed for this purpose, and is not utilized well. Unified Memory can provide virtual larger memory than the physical memory by just changing the allocation method, but the execution speed is the slow. In this study, we appended APIs to control behavior of Unified Memory to a numeric library cupy, that is used for Chainer, and developed a test program to investigate CUDA runtime's behavior for Unified Memory, and investigated the behavior on Power9 Processor and VOLTA CPU. We also discuss effective implementation of Deep Learning frameworks using Unified Memory according the investigation results.

Keywords: Deep Learning, Large-scale neural network, Unified Memory, GPU, Chainer

1. はじめに

ディープラーニングは画像認識，音声認識など様々な分野で利用されており，年々その精度や処理速度が向上しており，より高い精度を得るために，モデルの大規模化が進んでいる．大規模画像認識大会 ILSSVC では 2012 年に 8 層の AlexNet[1]が優勝したが，2015 年には 152 層の ResNet[2]が優勝しており，ネットワーク全般で層やパラメータの数が増加している[3]．医療分野でより精度の高い診断を行うためなどにより解像度の高い画像や 3D 画像等，データ量が大きいデータを直接ディープラーニングで処理する事が増えており，モデルや処理中のデータを保持するためのメモリ量が飛躍的に増加している．

一方，ディープラーニングの処理で多くの場合用いられる GPU の物理メモリ量は計算能力の向上と比較すると十分に拡大していない．例えば，Nvidia 社の前世代 GPU PASCAL と現世代 GPU VOLTA を比較すると，演算速度は単精度(32bit)で 6 倍，半精度では(16bit)で 12 倍になってい

†1 日本アイ・ビー・エム株式会社 東京基礎研究所
IBM Research - Tokyo

るのに対し，メモリ量は 16 GB から 32 GB の 2 倍に留まっている．GPU 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．

この様な状況ではディープラーニングの処理に必要なメモリが GPU の物理メモリを超えることが十分に考えられる．使用メモリ量は「ミニバッチ」と呼ばれる一回に処理する入力データの個数の調整によってある程度制御可能だが，近い将来ミニバッチを最小の 1 としても GPU の物理メモリ以上のメモリが必要となることも考えられる．

GPU プログラミングで仮想メモリは使用可能だが，実メモリ以上の仮想メモリを使用することはできない．このため，フレームワークの開発者がメモリを明示的に CPU メモリに一時的に退避し必要時 GPU メモリに回復したり，再計算可能な内容を含むメモリを一時的に削除して必要時に再計算することにより大規模モデルのサポートを実現しているが，これはフレームワークの複雑化をもたらしその開発生産性を低下させる．また，フレームワークの利用者もプログラムのオプションやモデルの設計，パラメータを実メモリの大きさを意識して調整している．このように大規模モデル使用に際してフレームワークの開発者や利用者は使用メモリを実メモリ以下とするために生産性を低下させており，今後もこの課題はより重要になると考えられる．

GPU 上でプログラミングは，CUDA[4]と呼ばれる Nvidia 社が提供するプログラミング環境で行われることが多い．CUDA では Unified Memory[5]と呼ばれる仮想メモリ機能が提供されている．Unified Memory は PC と GPU でメモリを透過的に共有することを主な目的として提供された機能である．メモリ確保用 API の変更のみでプログラマから透過的に GPU での実メモリ以上のメモリアクセスを可能にする．この機能により上記生産性低下の課題を解決できる可能性がある．

Unified Memory を GPU の使用メモリ拡大に用いる際の課題は性能である．Chainer[6][7] Version2, PASCAL GPU, POWER8 を用いたこれまでの測定[8]で Unified Memory により実メモリの数十倍の大きさのモデルを実行可能であるが，その際の実行速度は入力量あたり最大 1/10 程度まで低下することが判明している．実行速度は CUDA ランタイムによる Unified Memory のメモリ退避・回復の挙動により大きく変動するが，その挙動は公開されていない．現在使用可能な VOLTA は PASCAL が持たなかったページアクセスカウンタを持つため，Unified Memory 機能の振る舞いが大きく変更されている可能性がある．

本稿では，Chainer Version4, VOLTA GPU, POWER9 上で Unified Memory 用テストプログラムを用いて，Unified Memory の動作を検証した．また，検証結果に基づいてディープラーニングフレームワークを効率的に実装する方法について考察した．以下，第 2 章では大規模モデルサポートのための技術について，第 3 章では VOLTA GPU, POWER9 による Unified Memory 機能の検証について，第 4 章では Unified Memory を用いたディープラーニングフレームワーク特に Chainer における実装方針について考察し，第 5 章でまとめを述べる．

2. 大規模モデルサポートのための技術

医療分野[9]を中心に大規模モデルへの要求が高まっている．例えば肺ガン診断でより精度の高い診断を行うためにはより解像度の高い 3D 画像を処理する必要がある．問題によっては画像を分割して複数のディープラーニングモデルで処理することより可能な場合もあるが，局所的な画像(癌候補の形状)と俯瞰画像との関連（肺中の位置や血管との位置関係）を同時に判断するには全てのデータを一つのモデルで処理することが望ましい．このような場合，従来は複数の角度や解像度の断面画像を一つのモデルに同時に入力する等の方法により入力データサイズを削減してディープラーニングで処理を行っていたが，3D 画像をそのまま処理することができればより正確な診断が期待できる．また一般に同じサイズの入力データであっても精度向上のために層やパラメータ数は増加する傾向にあり，精度向上の観点からも処理に必要なメモリサイズは増加している．

本章では，このような大規模モデルを処理するための技術を概観するが，説明の前に本稿で想定する「ディープラーニング」の処理手順について説明する．なお，本稿ではディープラーニングで多く用いられる誤差逆伝播法を用いるニューラルネットワークを想定する．

2.1 ディープラーニングの処理手順

ディープラーニングでは通常複数の入力データをまとめて処理する．一回で処理するデータの個数を「ミニバッチ」，一組のデータの処理を 1 イテレーションと呼ぶ．

図 1 に L 層のニューラルネットワークの誤差逆伝搬法における処理とデータの流れを示す．上段に順伝搬処理，下段に逆伝搬処理が書かれている．

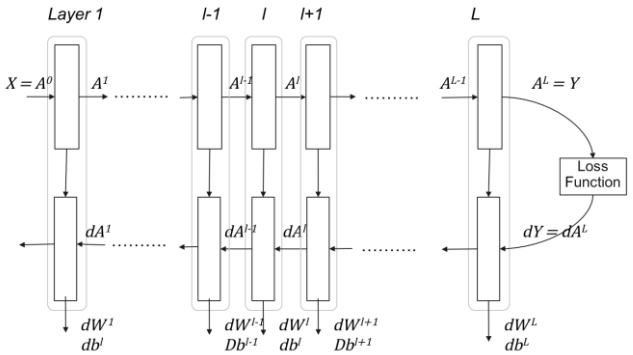

図 1：各イテレーションの処理とデータの流れ

順伝搬処理の l 層においては，l-1 層の出力 $A[l-1]$を入力とし，重み $W[l]$ とバイアス $b[l]$から(1)式を用いて，出力 $Z[l]$ を求める(図 2 参照).

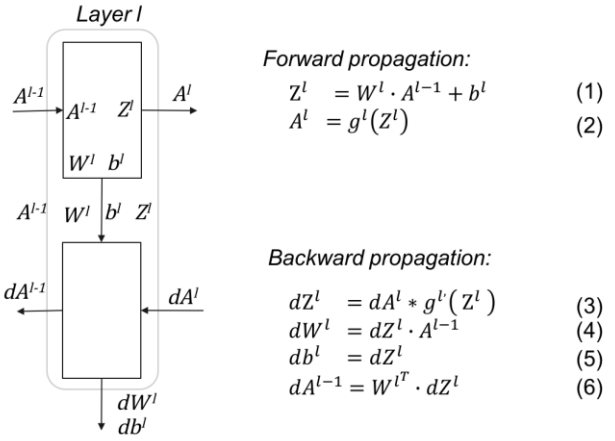

図 2: 各層の処理とデータの流れ

Z[l]に(2)式のように活性化関数 g を適用し l+1 層の入力とする．1 から L 層まで同様の処理を行い，Loss 関数の出力 *dY(=dA[L])*を逆伝搬処理の入力とし，逆伝搬処理を行う．逆伝搬の l 層においては，l+1 層の出力 *dA[l]*，l 層の順伝搬で入力 *A[l-1],* 計算結果の *Z[l]*，重み *W[*l], バイアス *b[l]*を用いて，(3)式から(6)式を用いて，l 層の逆伝搬における出力 *dA[l-1]*，重み更新の勾配 *dW[l],* バイアス更新の勾配 *db[l]*を求める．この処理を各層で順次行うことで全ての層の *dW, db* を求め，これらを用いて，重み *W*, バイアス *b* を更新する．この順伝搬処理，逆伝搬処理を 1 回のパラメータ更新としてこれを継続して行うことで Loss 関数の出力を小さくしていく．

以下，GPU メモリ使用量を検討するために，継続して保持する必要がある変数について考える．各層の計算中に必要な一時的な変数は高々一組しか同時に必要としないのでこの検討からは除外する．まず，学習対象である重み W 及びバイアス b はイテレーションに跨って保持する必要がある．また，順伝播の計算結果 Z と出力 A は逆伝播の対応する層で必要となる．多くのフレームワークでは，順伝播の終了後逆伝播の対応する層の処理まで順伝播の出力結果 Z を保持し，活性化関数の処理は軽いので出力 A は逆伝播処理時に出力結果 Z と活性化関数 g から再計算するため継続した保持は不要である．つまり，継続して保持する必要がある変数は，重み W，バイアス b，出力結果 Z である．重み W とバイアス b は 1 組分のみ必要だが，順伝播の計算結果はミニバッチの個数分用意する必要があるため，ミニバッチと比例して拡大する．各変数の保持に必要となるメモリの大きさは層の種類や入力データの大きさによって異なるが，一般に Convolution 層の場合出力結果 Z の変数のための領域が支配的となり，全結合層の場合重み W のための領域が支配的となる．GoogLeNet 等の典型的な Convolution Network ではほとんどの層が Convolution 層なので，出力結果 Z を保持するための領域が支配的となる．

2.2 再計算によるメモリ削減手法

この節から，大規模モデルサポートのための技術について説明する．この手法は，逆伝播処理時に必要だが再計算可能な変数を順伝播処理後に一旦削除し，逆伝播処理時に再計算することにより使用する GPU メモリを削減する手法[10]である．ディープラーニングでは通常 GPU の計算資源はほぼ 100%使用可能なので，再計算に必要なコストはそのまま計算時間の増加として現れる．削減可能な変数は，Convolution 層で支配的となる出力結果 Z のみで，重み W やバイアス b は再計算不可能なので削減できない．

2.3 Out-of-Core によるメモリ削減手法

この手法は順伝播処理終了後，変数を GPU メモリから PC 側メモリに退避し，逆伝播処理で対応する層が実行されるまでに PC 側メモリから GPU メモリに回復することにより，GPU メモリの使用量を削減する手法[11][12][13]である．変数の退避・回復のための CPU-GPU 間のメモリ転送は計算と並列に実行可能であるため，計算時間の方がメモリ転送時間よりも長ければ少ないオーバヘッドで導入可能である．

この手法では Convolution 層で支配的となる出力結果 Z だけでなく，全結合層で支配的となる重み W やバイアス b についてもメモリ削減の対象とすることが可能である．ただし，単一の層のみで GPU メモリ量を超えるメモリを必要とする場合には対応できない．

2.4 Unified Memory によるメモリ削減手法

CUDA Unified Memory と呼ばれる仮想メモリ機能では，CUDA ランタイムがプログラマから透過的に不要なメモリを CPU に退避し必要なメモリを GPU に回復することにより，GPU での実メモリ以上のメモリアクセスを可能にする．

この手法では全てのメモリを削減の対象とすることが可能である．また，単一の層のみで GPU メモリ量を超えるメモリを必要とする場合にも対応可能である．

本手法で問題となるのは実行速度である．この手法では，メモリの退避・回復を CUDA ランタイムに任せるため，フレームワークから対象メモリやタイミングを制御することが難しい．効率的に動作するには，フレームワークと CUDA ランタイムが適切に協調して動作する必要がある．

以下，本稿では Unified Memory によるメモリ削減手法の実行速度に着目して議論を進める．

3. Unified Memory 機能の検証

Unified Memory によるメモリ削減手法では Unified Memory に対する CUDA ランタイムの挙動が性能に大きく影響を与える．API や基本的な動作原理についての情報は公開されているが，CUDA ランタイムの具体的な挙動については公開されていない．本章では Power9 及び VOLTA GPU 上でテストプログラムを用いて CUDA9 の Unified Memory の挙動を検証する．CPU として intel 社の X86 プロセッサを使用した場合でも，Unified Memory に対する API や基本的な動作原理は変わらない．CUDA ランタイムの挙

動も大きな違いはないと考えるがX86プロセッサでの動作は検証しておらず今後の課題である.

3.1 Unified Memory の基本動作

Unified Memory に対する CUDA ランタイムの基本動作は以下の通り.

(1) **CudaMemAllocManaged:** Unified Memory 確保用 API. 引数は通常のメモリ確保用の CudaMemAlloc と同一である. メモリ確保時点では実メモリは割当てず, 仮想アドレスのみを割当てる. メモリの開放時は通常のメモリの開放と同一の CudaMemFree API を使用する.

(2) **CudaMemAdvise:** 指定した Unified Memory の領域のアクセス特性に関するアドバイスを与える API. アドバイスの種類については後述.

(3) **CudaMemPrefetchAsync:** 指定した Unified Memory の領域を非同期に GPU もしくは CPU に転送する API.

(4) **メモリアクセス時:** アクセス時点で GPU の実メモリが未割当てであればページフォールトが発生し, 実メモリを割当てる. メモリ内容が CPU に退避されていれば GPU メモリへに転送する. 連続領域へのアクセス等次のメモリアクセスが予測可能な場合は実際のアクセス前に GPU メモリへの転送を行う場合もある.

(5) **メモリ不足時:** CUDA ランタイムが GPU メモリを選択して CPU に退避する. 退避対象メモリ領域の選択方法については後述する.

3.2 Unified Memory へのアドバイス

CudaMemAdvise の引数として与えられるアドバイスの種類は以下の通り.

(1) **CudaMemAdviseSetReadMostly:** 指定デバイス(CPU もしくは GPU)において指定領域へのアクセスのほとんどが Read のみであることを指定. CUDA ランタイムは ReadMostly が指定されたデバイスにキャッシュを置くため高速な Read が可能になる. Write も可能だがコストは高い.

(2) **CudaMemAdviseSetPreferredLocation:** CUDA ランタイムに指定領域を指定デバイスに配置することが望ましいことを宣言.

(3) **CudaMemAdeviseSetAccessedBy:** CUDA ランタイムに指定領域を指定デバイスに配置することを指示. これにより指定デバイスに実メモリが割当てられ指定デバイスからのメモリアクセス時のページフォールトが発生しない.

3.3 退避対象メモリ領域の選択方法

公開されている退避対象メモリ領域の選択方法は以下の通り.

(1) CUDA ドライバは GPU メモリ領域を優先殿順に並べたキュー（以下退避キューと呼ぶ）を一本持つ.

(2) 退避キュー中でメモリ領域は LRU(Least Recently Used) 順に並べられ先頭の領域が最初に退避の対象となる.

(3) メモリ領域は新規に確保されるか対象デバイスに転送されると「使用中」とみなされる.

以上が公開されている退避対象メモリの選択方法であり, 特にメモリの削除, アドバイス, Prefetch 等があった場合に退避キューがどう制御されるか等の具体的な選択手順については公開されていない.

Unified Memory を効率的に使用するには, メモリアクセスパターンに合わせて CPU-GPU 間でメモリ転送を行う必要がある. この内 CPU から GPU へのメモリ転送に関しては CudaMemPrefetchAsync API によりある程度制御可能であるため, 以下, CUDA ランタイムによるメモリ不足時のメモリの空きを作るための退避対象となるメモリ領域の選択方法に着目して動作を検証する.

3.4 退避対象メモリ領域の選択手順の検証アルゴリズム

以下メモリ不足時の CUDA ランタイム挙動検証のためのアルゴリズムを示す.

(1) **Prepare:** Unified Memory を一定単位(例えば 1GB)毎に N 単位確保し, GPU のほぼ全ての実メモリ(例えば 16GB のメモリを持つ VOLTA GPU であれば 14 単位)を確保する. 確保の際全ての領域にアクセスし GPU の実メモリを割当てる.

(2) **Action:** <退避キューの優先度に影響を与える可能性のある操作>をする.

(3) **Overcommit:** Unified Memory を更に一単位確保し, overcommit の状態にする.

(4) **Touch:** 最初に確保した Unified Memory の領域にアクセスして, アクセス時間を観察する. アクセス時間は GPU 上に有効な実メモリがあれば数百ミリ秒, なければ数十秒となるので, この時間により対象領域に対して対象領域が CPU に退避されたか否かを判定できる.

Action 操作が退避キューの操作に影響を及ぼさなかった場合, Overcommit 最初に確保した領域が選択され退避される. もし最初に確保した領域が退避されていなければ Action 操作が退避キューの優先度に影響を及ぼしたことになる. 以下本論文では退避キューの優先度を退避優先度と呼ぶ. 次回メモリが不足した場合退避優先度が最も高い領域がの退避対象となる.

今回の実験では, 対象領域以外の領域への影響を確認するために Touch 操作で最初に確保した領域へのアクセスに加えて, 2 番目, 3 番目に確保した領域へのアクセスについても実行時間を計測した.

3.5 検証項目

本節では, 以下の操作を退避優先度を制御する操作の候補として動作を検証した. これらの操作は, 上記アルゴリズムの Action 操作で使用する.

(1) 検証対象領域にメモリアクセスする.

(2) 検証対象領域を GPU に Prefetch する.

(3) 複数の検証対象領域を GPU に Prefetch する.

(4) 検証対象領域より退避優先度が低い領域を開放してメモリの空き領域を拡大し上記アルゴリズム(2)でovercomitが起きないようにする．
(5) 検証対象領域より退避優先度が低い領域を CPU に Prefetch する．（c.f. これにより退避優先度が上がることを期待）
(6) (4)と同様に検証対象領域より退避優先度の低いメモリを開放し，overcomitが起きないようにした後に，検証対象領域を CPU に Prefetch する．（c.f. これにより GPU 側実メモリ割当てが保持されるかを検証）
(7) 検証対象領域に CudaMemAdvise API により CudaMemSetAccessedBy で GPU メモリへの配置を指定．
(8) 検証対象領域に CudaMemAdvise API により CudaMemAdviseSetPreferredLocation で GPU メモリへの配置が望ましいことを宣言する．
(9) 検証対象より退避優先度が低い領域に CudaMemAdvise API により CudaMemSetAccessedBy で CPU メモリへの配置を指定する．(c.f. これにより退避優先度が上がることを期待)
(10) 検証対象より退避優先度が低い領域に CudaMemAdvise API により CudaMemAdviseSetPreferredLocation で GPU メモリへの配置が望ましいことを宣言する．(c.f. これにより退避優先度が上がることを期待)

3.6 検証プログラム

今回，Chainer で使用される GPU 上での数値計算のための Python 拡張モジュールである cupy[14] (version4.0.0b2)に Unified Memory を制御する機能を追加し，この機能を使って上記アルゴリズムを実装した．なお，Cupy のメモリ管理では，メモリの確保・開放処理のオーバヘッドを削減するために一旦確保したメモリは開放せずに Memory Pool と呼ばれるテーブルに保存し再使用する機能があるが，Unified Memory 使用時には悪影響があるため今回の検証では使用しなかった．追加した API は以下の通り．

(1) cupy.Prefetch(self, stream, deviceid=None): CudaMemPrefetchAsync により cupy オブジェクトを指定デバイスに Prefetch する API
(2) cupy.advise(self, int advise, cupy.cuda.Device device): CudaMemAdvise API により cupy オブジェクトにアドバイスを与える API

以下，検証プログラムの擬似コードを示す．

```
# allocate chunks almost full
x = []
for i in range(0, <# of chunks>):
  newx = xp.zeros(size * (1024 * 1024),
                  dtype=numpy.float32)
  if hint > 0:
    newx.advise(hint, device)
  x.append(newx)
# do something to change the priority
action(xp, args.action, x, 0, device, args.gpu)
# allocate one more chunk for overcommit
newx = xp.zeros(size * (1024 * 1024),
                dtype=numpy.float32)
if hint > 0:
    newx.advise(hint, device)
# touch the first chunk
cupy.cuda.runtime.deviceSynchronize()
tstart = time.time()
xp.dot(x[unit], x[unit].T)
cupy.cuda.runtime.deviceSynchronize()
tend = time.time()
print('TIME=',  tend- tstart)
```

また各検証項目に相当する操作の擬似コードを示す

```
def action(xp, action, x, unit, device, gpu):
    if action == 1:
        return touch(xp, x, unit)
    elif action == 2:
        strm = cuda.Stream(non_blocking=True)
        x[unit].Prefetch(strm)
    elif action == 3:
        strm = cuda.Stream(non_blocking=True)
        x[unit].Prefetch(strm)
        x[unit+1].Prefetch(strm)
    elif action == 4:
        v = x[unit]
        x[unit+3] = None
        del v
    elif action == 5:
        strm = cuda.Stream(non_blocking=True)
        x[unit+1].Prefetch(strm, device=-1)
    elif action == 6:
        v = x[unit]
        x[unit+3] = None
        del v
        strm = cuda.Stream(non_blocking=True)
        x[unit].Prefetch(strm, device=-1)
    elif action == 7:
        x[unit].advise(5, device) #
cudaMemAdviseAccessedBy=5
    elif action == 8:
        x[unit].advise(3, device) #
cudaMemAdvisePreferredLocation=3
    elif action == 9:
        x[unit + 1].advise(5,
chainer.cuda.get_device_from_id(-1))
    elif action == 10:
        x[unit + 1].advise(3,
chainer.cuda.get_device_from_id(-1))
```

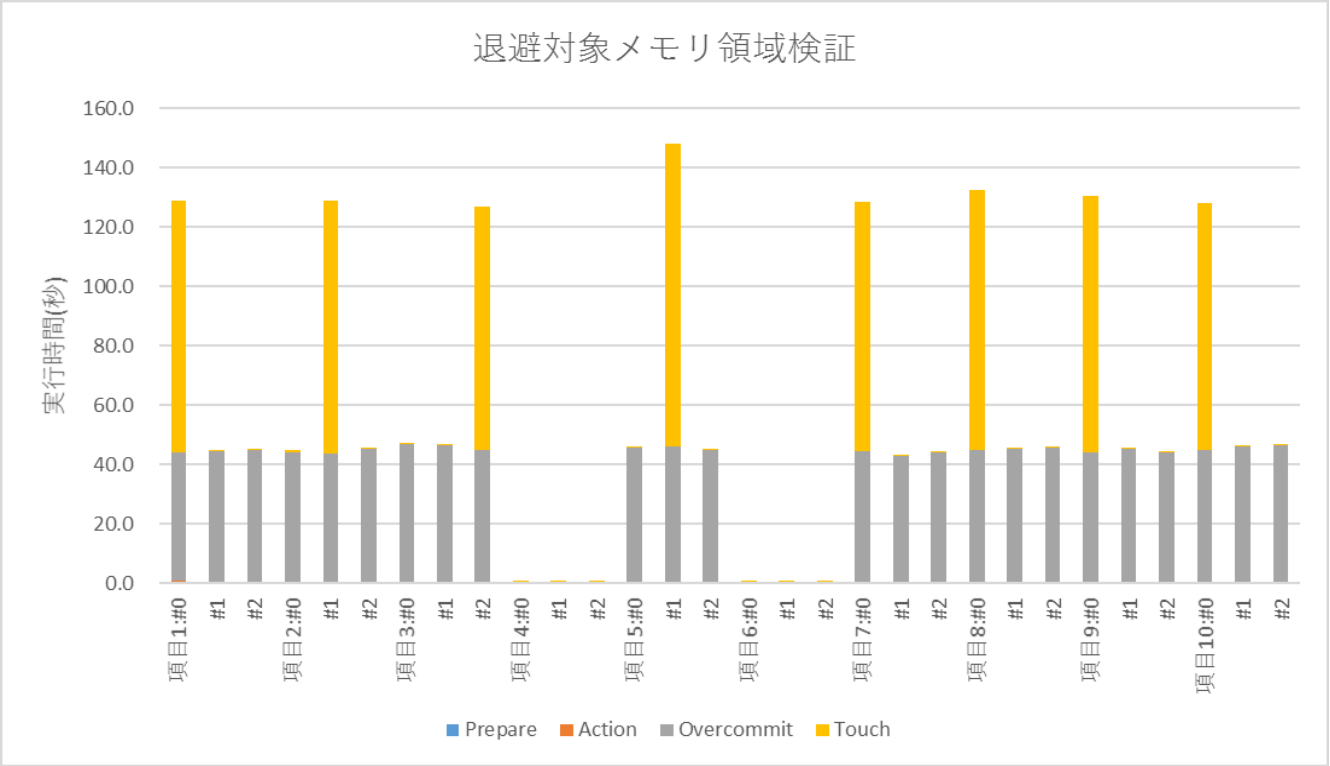

図 3: 退避対象メモリ領域プログラム実行時間

3.7 検証環境

計測マシンには IBM® Power Systems™ AC922(通称 Newell)[15]を用いた．Newell は CPU に POWER9 プロセッサー(10 コア，2.0GHz)を 2 基搭載し，CPU メモリー1TB であり，2 ソケットの NUMA 構成となっている．GPU は NVIDIA® Tesla® V100(GPU メモリー16GB)を 4 基搭載している．CPU-GPU 間は NVLink2.0(片方向 25GB/sec のバンド幅)で接続されている．

ソフトウェアには，Ubuntu-16.04, CUDA9.1, Python 3.6.0, Cupy4.0.0b2 を使用した．

3.8 各項目検証結果

図 3 に 3.5 節の各検証項目毎に 3.4 節の各操作の実行時間の累計を示す．図中#0, #1, #2 は 3.4 節の Touch 操作時のアクセス対象(最初，2 番目，3 番目)を示す．

測定結果及び考察は以下の通り．

(1) 検証対象領域にメモリアクセス．
➔ 退避優先度に変化なし．VOLTA GPU にはページアクセスカウンターが新設されたがメモリアクセスでは優先度は変化しない．

(2) 検証対象領域を GPU に Prefetch する．
➔ Prefetch により対象領域の退避優先度最低となり，GPU メモリからの退避が回避される．図 3 の項目 2 のグラフから，最初に確保した領域に代わって 2 番目に確保した領域が退避されていたことが確認できる．

(3) 複数の検証対象領域を GPU に Prefetch する．
➔ 複数の領域の優先度ががされ，それに準じる領域が繰り上がって CPU メモリに退避される．図 3 の項目 2 のグラフから，最初に確保した領域に代わって 3 番目に確保した領域が退避されていたことが確認できる．

(4) 検証対象領域より退避優先度が低い領域を開放してメモリの空き領域を拡大し上記アルゴリズム(2)で overcomit が起きないようにする．
➔ 全ての領域について CPU メモリへの退避が行われない．Overcommit の時間もなくなり overcommit が起きていないことが確認できる．

(5) 検証対象領域より退避優先度が低い領域(2 番目に確保した領域)を CPU に Prefetch する．
➔ CPU への Prefetch により優先度の低かった領域の退避優先度が最優先となり最初に退避された．検証対象領域の退避は回避された．図 3 の項目 5 のグラフから，2 番目に確保した領域が退避されていたことが確認できる．

(6) (4)と同様に検証対象領域より退避優先度の低いメモリを開放し，overcomit が起きないようにした後に，検証対象領域を CPU に Prefetch する．
➔ 対象領域を CPU に Prefetch しても GPU 側実メモリ割当ては保持された．Overcommit の時間もなくなり overcommit が起きていないことが確認できる．

(7) 検証対象領域に CudaMemAdvise API により CudaMemSetAccessedBy で GPU メモリへの配置を指定．
➔ 退避優先度に変化なし．

(8) 検証対象領域に CudaMemAdvise API により CudaMemAdviseSetPreferredLocation で GPU メモリへ

の配置が望ましいことを宣言
➔ 退避優先度に変化なし.

(9) 検証対象より退避優先度が低い領域に CudaMemAdvise API により CudaMemSetAccessedBy で CPU メモリへの配置を指定.
➔ 退避優先度に変化なし.

(10) 検証対象より退避優先度が低い領域に CudaMemAdvise API により CudaMemAdviseSetPreferredLocation で GPU メモリへの配置が望ましいことを宣言.
➔ 退避優先度に変化なし.

3.9 検証結果に対する考察

検証項目に対する考察は以下の通り．ただし，この考察結果は現在の CUDA の実装に対するものであり，今後のバージョンアップや測定条件の違いで変わる可能性があることに注意が必要である.

(1) メモリアクセスでは退避優先度は変化しない. なんらかの CUDA API の呼出しが必要.

(2) 指定領域の GPU への CUDA MemPrefetchAsync 操作により退避優先度を非同期で最高にすることが可能. ただし，対象領域が GPU 上にない場合 CPU から GPU への転送命令が CUDA Stream の実行キューに追加される. 転送前に対象領域が不要になった場合でもこの命令はキャンセルされず，転送終了後に開放される.

(3) 指定領域の CPU への CudaMemPrefetchAsync 操作により退避優先度を非同期で最低にすることが可能. ただし，退避領域が CPU 上にない場合 GPU から CPU への転送命令が CUDA Stream の実行キューに追加される. 転送前に対象領域が不要になった場合でもこの命令はキャンセルされず，転送終了後に開放される.

(4) Cuda MemAdvise によるアドバイスは対象領域へのアクセスの際の Page Fault の削減には有効だが，退避優先度を変化させることはない.

4. Unified Memory を用いたディープラーニングの実装方針

本章では，前章での考察に基づいて CUDA で使用可能な API を使用した効率的なディープラーニングフレームワーク実装方針について検討する.

4.1 基本実装方針

Unified Memory を使用してディープラーニングフレームワークを実装する場合，以下の 3 つの基本方針が考えられる. (1)から(3)に進むにつれ大規模モデルをサポートするために必要なフレームワークの修正点が少なくできる.

(1) **直接指示方式:** 通常メモリの代わりに Unified Memory を使用する．フレームワークが実メモリの量を，GPU-GPU 間のメモリ転送を直接指示し，必要な GPU 実メモリが常に GPU の実メモリ以下になるように制御する．2.3 節で説明した Out-of-Core 方式とフレームワークが転送を指示する点が共通しているが，Out-of-Core 方式では通常のメモリを本方式では Unified Memory を用いる.

(2) **分業方式:** フレームワークは状況に応じてメモリ退避優先度を制御するが実メモリの量には感知しない. CUDA ランタイムは退避優先度に従って，使用可能な実メモリの量の変動に対応して CPU-GPU 間でメモリ転送を行い，GPU メモリを管理する.

(3) **CUDA ランタイム全面依存方式:** 大規模モデルサポートのための Unified Memory の管理を全て CUDA ランタイムに任せる.

4.2 各実装方式の実現可能性とメリット・デメリットについての考察

以下，この 3 つの基本方針が 3.9 節で検討した CUDA Unified Memory の機能で実装可能であるかについて検討し，各方針のメリット・デメリットについて考察する.

(1) **直接指示方式:** Unified Memory は CPU-GPU 間の CudaMemPrefetchAsync というメモリ転送 API があり

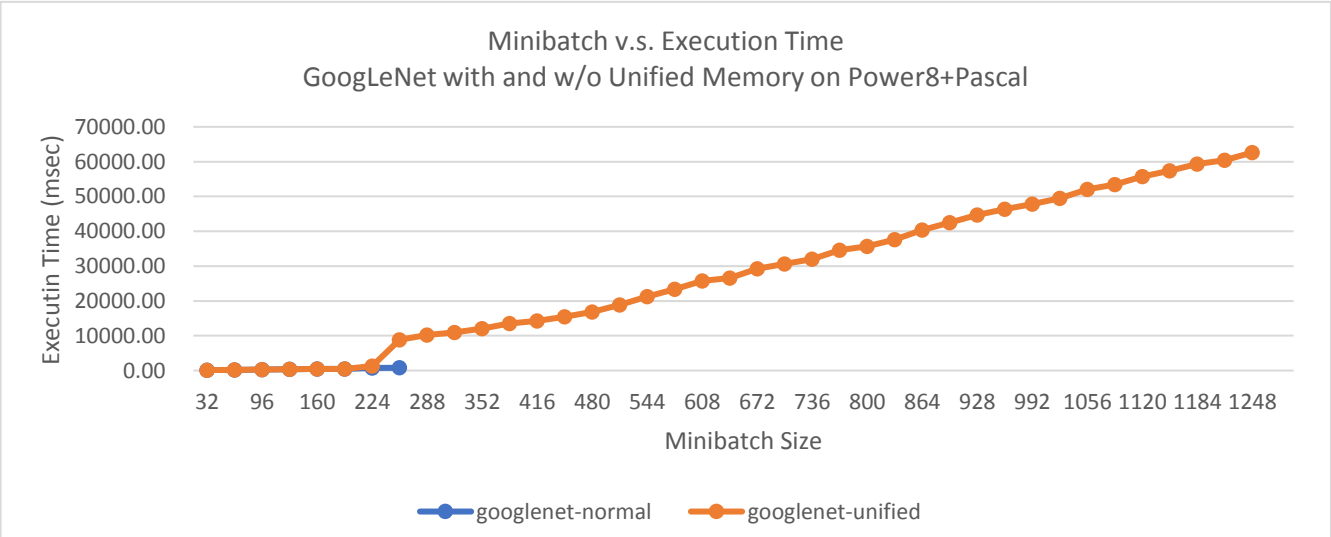

図 4: 通常メモリと Unified Memory の性能比較

これにより指示可能である．3.9 節の通り CPU から GPU へのメモリ転送後対象領域の退避優先度が最低に，GPU から CPU へのメモリ転送後対象領域の退避優先度が最高になるため，フレームワークが GPU の使用メモリを実メモリ量以下に制御していれば使用中の領域が CPU に退避されることはない．図 4 は Power8 プロセッサと VOLTA GPU 上で Unified Memory を使用した場合と通常メモリを使用した場合の実行速度比較である．Unified Memory を使用しても使用メモリが GPU の実メモリ以下であれば通常メモリの場合と同等の実行速度を得られるため，本方式でも Ouf-of-Core 方式と同等の実行速度を得られる．本方式では Out-of-Core 方式と異なり動作するが性能が得られない状態から実装を始め順次性能を改善しながら実装を進めることが可能であるので，Out-of-Core 方式よりもフレームワークの大規模メモリサポートの開発生産性は高い．

(2) **分業方式:** 3.9 節に非同期で退避優先度を変更する操作が示されているので，これら操作を用いて実装することは可能．ただし，退避優先度は処理が進む毎に随時変更されるが上記操作によって発行されたメモリ転送命令は取り消すことができないので十分な性能が得られないことが予想される．

(3) **CUDA ランタイム全面依存方式:** 図 4 に示した Unified メモリの性能[8]は CUDA ランタイム全面依存方式によるものだが Unified Memory を使用した場合の性能は十分とは言えない．CUDA ランタイム全面依存方式で十分な性能を得るにはメモリアクセスによる退避優先度の変更が必要と考えるが，現在の測定環境ではサポートされていないことが確認できた．今後のアップデートでサポートされれば性能や問題点ついて検討したい．

5. まとめと今後の課題

本稿では，Chainer で使用する数値演算ライブラリ Cupy に Unified Memory を制御する API を追加し，Unified Memory の振る舞いを調査する検証プログラムを作成した．また，作成した Power9 及び VOLTA GPU 上で動作を検証した．その調査結果に基づいて，Unified Memory を用いてディープラーニングフレームワークを効率的に実装する方法について考察した．各実装方式のメリット・デメリットについて考察し，現在の Unified Memory の機能を用いる場合，直接指示方式であれば Out-of-Core 方式と同等の性能を達成可能であるが，現状の Unified Memory では分業方式での実装は困難であることを示した．

今後の課題は X86 プロセッサでも同様の検証を行うことと，検討した実装方針に基づいて実際のフレームワークで実現し，性能評価を行うことである．

参考文献

[1] Alex Krizhevsky, Ilya Sutskever, and Geo rey E. Hinton. 2012. ImageNet Classification with Deep Convolutional Neural Networks. In International Conference on Neural Information Processing Systems. 1097–1105.
[2] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. 2015. Deep Residual Learning for Image Recognition. CoRR abs/1512.03385 (2015). http://arxiv.org/ abs/1512.03385
[3] Christian Szegedy, Wei Liu, Yangqing Jia, Pierre Sermanet, Scott E. Reed, Dragomir Anguelov, Dumitru Erhan, Vincent Vanhoucke, and Andrew Rabi- novich. 2015. Going deeper with convolutions. In IEEE Conference on Computer Vision and Pattern Recognition. 1–9.
[4] NVIDIA, CUDA Toolkit, https://developer.nvidia.com/cuda-toolkit
[5] NVIDIA, Unified Memory on Pascal and Volta, http://on-demand.gputechconf.com/gtc/2017/presentation/s7285-nikolay-sakharnykh-unified-memory-on-pascal-and-volta.pdf
[6] Preferred Networks, Chainer, https://chainer.org/
[7] Preferred Networks, Chainer github, https://github.com/chainer/chainer
[8] Yasushi Negishi, Haruki Imai, Jun Doi, Kiyokuni Kawachiya,"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," (in Japanese), 日本ソフトウェア科学会第 34 回大会, Sepember 19-21, 2017.
[9] Jose Dolz, Christian Desrosiers, Ismail Ben Ayed, 3D fully convolutional networks for subcortical segmentation in MRI: A large-scale study, In NeuroImage, 2017, , ISSN 1053-8119, https://doi.org/10.1016/j.neuroimage.2017.04.039.
[10] Tim Salimans and Yaroslav Bulatov, Saving memory using gradient-checkpointing, https://github.com/openai/gradient-checkpointing
[11] Haruki Imai, Tung Le Duc, Taro Sekiyama, Kiyokuni Kawachiya, "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," (in Japanese), 第 162 回 PHC 研究発表会, SIG HPC of IPSJ, December 18-19, 2017.
[12] Chainer version 2 Out-of-Core 学習用派生レポジトリー: https://github.com/anaruse/chainer/tree/OOC_chainer_v202
[13] Cupy version 1 Out-of-Core 学習用派生レポジトリ: https://github.com/anaruse/cupy/tree/OOC_cupy_v102
[14] Preferred Networks, cupy github, https://github.com/cupy/cupy
[15] IBM Power System AC922 https://www.ibm.com/jp-ja/marketplace/power-systems-ac922