Vol.2021-DPS-186 No.65
Vol.2021-CSEC-92 No.65
2021/3/16

推論多重実行における GPU 資源利用効率化技術

鈴木貴久[1]　田中美帆[1]　豊永慎也[1]　松倉隆一[1]

概要：AI（Deep Learning）技術の普及とハードウェアの進歩により，一台のデバイス上で複数の推論処理を行うようになってきている．これにより，1 台の GPU を複数の推論処理で共有して利用できるようになるが，一方で共有による資源の競合や重複が発生し利用効率が低下する．そのために，複数の推論処理での GPU 資源利用を管理・制御し，資源利用を効率化する機構が必要となる．従来でも，GPU の処理単位で複数の処理を同時実行する技術はあったが，一般に推論ではデータをロードして，推論して，出力するため，推論中に別の処理に干渉されると処理時間が増加してしまう．そのため，本稿では推論の単位で資源利用を管理・制御することで GPU 利用を効率化する技術を提案する．また，実際に推論を使った画像処理のアプリケーションでその効果を評価し，1.6 倍の効率化を達成した．

キーワード：AI，Deep Learning，推論，GPU，並列処理

Efficient GPU Resource Management for Multiple AI Inference Processes Execution

TAKAHISA SUZUKI[†1]　MIHO TANAKA[†1]
SHINYA TOYONAGA[†1]　RYUICHI MATSUKURA[†1]

Abstract: With the spread of artificial intelligence (Deep Learning) technologies and advances in hardware technologies, multiple inference processes are becoming to be performed on a single device. As a result, a single GPU would be shared by multiple inference processes. But due to sharing one GPU, its utilization efficiency is decreased by resource contention and duplication. Therefore, a mechanism to manage and control GPU resource utilization in multiple inference processing is required to make resource utilization efficient. Conventionally, there has been a technique which executes multiple processes in parallel on one GPU. On the other hand, generally, an inference process is consisted of, a data load, a perform an inference, and an output result phases, but if another process make an interfere during the inference phase, the processing time would increase. Therefore, in this paper, we propose a technique to improve the efficiency of GPU utilization by managing and controlling resource utilization in units of inference process. And, the effect was evaluated by an image processing application which uses an inference, and the efficiency improvement of 1.6 times was achieved.

Keywords: AI, Deep Learning, Inference, GPU, Parallel Processing

1. はじめに

近年，Deep Learning に代表される AI 技術が普及してきており，様々な分野で活用され始めている．一例として，映像処理の分野では ResNet[1]や YOLO[2]に代表される画像に映っている物体を認識ならびに検知する手法が挙げられる．これらを利用することで映像監視を自動化し，防犯・防災等に役立てることが期待されている．AI 技術による物体検知等の推論処理を利用することで，従来の人間の目による監視では現実的ではないほどの大量の映像の監視が可能になる．

推論処理には膨大な計算能力が必要となるため，近年では畳み込みニューラルネットワークなど，計算量の大きな推論処理の中核部分を GPU により処理することで推論処理を高速に行うケースが増えてきている．一方で，GPU の処理能力にあったデータサイズやアルゴリズム，推論処理の頻度を選択することは難しく，このようなアプリケーションの要件に合わせて GPU を選択し，単独のアプリケーションで GPU を占有利用するケースが多い．推論処理に利用できる GPU はまだまだ高価であるため，単独での利用が高コスト化に繋がっている．

そのため，推論処理のさらなる普及には，GPU 資源の利用効率を高めて相対的に処理コストを低減することが求められる．本稿では，GPU 資源を効率的に利用する方法として，GPU に入力する処理量をアプリケーションの要件に基づいて適切に制御することと，GPU 内の処理を並列化して処理の空き時間を最小にすることで，高価な GPU を複数のアプリケーションから効率的に利用することを可能にし，推論処理の低コスト化を実現することを目標としている．

2. 関連研究

カメラ画像の推論処理において，1 台の GPU で複数の推論処理を並列に実行し，GPU の利用効率を高める手法が提案されている．大きく分けると 2 つの方法があり，1 つは複数の入力画像を 1 枚の画像に変換して，1 つの処理として推論処理を行うバッチ処理[3]と呼ばれる方法で，もう 1

1 (株)富士通研究所
Fujitsu Laboratories LTD.

つは複数の入力画像を 1 台の GPU で並列に推論処理する方法である．

以降では，GPU を利用した推論処理の分野で広く普及している NVIDIA の GPU を利用する場合での，バッチ処理と並列処理について説明する．

2.1 バッチ処理

推論処理を GPU で実行するには，事前の学習で得られた学習モデルを GPU メモリに配置してから処理を行う．一般に学習モデルの大きさは数十 MB から数百 MB であり，非常に大きい．複数のカメラ画像に関する推論処理を行う場合，学習モデルは入力される画像毎に持つ必要があるため，たとえ同じ学習モデルを使用する場合でも，GPU のメモリ上に学習モデルを複数配置する必要がある．ただし，GPU のメモリ容量が不足して複数置くことができない場合は，学習モデルを随時入れ替えて推論処理を行う必要がある．これに対して，入力画像を 1 つの画像に纏めて推論処理を行うようにすれば，GPU メモリに配置する学習モデルも 1 つになり，GPU のメモリ効率を高めることができる．

GPU を利用して推論処理を行う場合，推論処理に必要な学習モデルと画像データを事前に GPU に転送し，GPU に対して推論処理の開始等を指示しなければならない．これらの API を提供するものが AI フレームワークであり，GPU を利用する場合には，この AI フレームワークを利用することが多い．今回は，バッチ処理をサポートする TensorRT[4]を利用する．TensorRT では大きさが M のデータ N 個を M×N の多次元データに変換し，一度の推論で複数のデータを一括で処理できるライブラリを提供している．入力画像サイズが大きくなるため，推論処理の時間は増加するが，入力画像毎に処理を実行するオーバーヘッドを削減でき，全体の推論処理の効率化が可能となる．

ただし，バッチ処理で集約可能なものは同じ学習モデルを使った推論処理のみに限られるという制約がある．

2.2 並列処理

バッチ処理は，GPU にカメラ画像が配置される前の処理であるが，GPU 内部での効率化が並列処理である．GPU には複数の演算器があり，複数の推論処理を並列に実行できる．GPU での推論処理に関わる資源には，演算器とメモリの 2 種類があり，並列処理を行うためにはこれらの資源を管理し，効率的に割り当てる必要がある．メモリの共有については CUDA[5]，演算器の共有については MPS（Multi Process Support）[6]が NVIDIA の GPGPU では提供されている．CUDA や MPS は，TensorRT と同様に GPU を利用するためのライブラリであるが，GPU の演算器やメモリ等の資源を直接制御することができる．一般的には，CUDA や MPS の機能は TensorRT 等の AI フレームワークから利用される．

CUDA では GPU のメモリを管理する API を提供しており，アプリケーションからはこの API を利用して GPU 側のメモリの確保・解放やデータの入出力を行う．この API は複数の推論処理の実行に対応しており，この API を利用することで複数の推論処理から GPU のメモリを安全に共有することができる．

CUDA では推論処理をカーネル[7]という単位で実行する．GPU には複数の演算器（CUDA コア）があり，カーネルを実行するには複数の CUDA コアを割り当てる．つまり，Deep Learning 等の推論処理では，内部で高度な並列処理を行っており，個々の CUDA コアは分解された単純な計算を繰り返し，全体として処理結果を統合しながら処理を進める．そのため，CUDA コアを利用して推論処理を行う場合には，事前に必要な CUDA コアを確保してから処理を開始することになる．複数の推論処理を並列処理する場合には，カーネルが排他的に CUDA コアを確保する．したがって，複数の推論処理に必要な CUDA コアが確保できない場合には，カーネル間で資源の取り合いになり，オーバーヘッドが増えて推論処理の開始から完了までのそれぞれの時間は増加してしまう．このため，スループットの面での向上は期待できない．

この問題を解決するのが，MPS 技術である．MPS では複数のカーネルで GPU の演算器を共有することが可能になっている． MPS では複数の CUDA コアの複数のカーネルへの割り当てを効率化し，複数のカーネルを並列に処理する．例えば，GPU が 100 個の CUDA コアを持っているときに，1 つのカーネルが 60 個の CUDA コアを必要だとすると，残り 40 個の CUDA コアを別のカーネルに割り当てることができる．

MPS による CUDA コアの共有方法には 2 種類ある．1 つは，カーネルへの CUDA の割り当てを MPS に任せる方法である．この場合，同時実行されるカーネルが要求する CUDA コア数と，GPU の持つ CUDA コア数から，MPS がカーネルへの CUDA コアの割り当てを決定する．もう 1 つは，全体のアプリケーションから各推論処理で利用可能な CUDA コア数の上限を設定する方法である．これにより他のアプリケーションのカーネルに邪魔されることがなくなり，所謂 QoS 制御のように推論開始から終了までの遅延時間を保証することが可能となる．

2.3 考察

従来技術を整理すると，バッチ処理のように GPU に渡す処理やデータを統合し，1 台の GPU で処理可能な処理サイズに適合される技術と，GPU での推論の並列処理を効率化する技術からなると言える．また，並列処理では，GPU のメモリ管理として，GPU に学習モデルや画像データを転送する機能と，Deep Learning 等の推論処理を実行する機能に分けることができる．

ここでは，並列処理の効率化を検討する上で必要となる実際の推論処理の挙動について説明する．例として SSD-

Vol.2021-DPS-186 No.65
Vol.2021-CSEC-92 No.65
2021/3/16

ResNet[8]をあげる. SSD-ResNet はカメラ画像から人や物を切り出し，これが何であるかを識別するアルゴリズムである． 図 1 は SSD-ResNet の構造を示す．図 1 にあるように，SSD-ResNet では，入力画像が左から入り，複数の処理を経て, 右から結果が出力される. 処理は 2 段階からなり，前段部分ではニューラルネットワークによる画像の特徴を抽出する．後段では，抽出された特徴から画像の領域を切り出し，そこに映っている人や物を検知した結果を出力する．前段のニューラルネットワークでは，画像の畳み込み演算をしながら，画像サイズを小さくしていく．分解された各処理では，CUDA コアが並列処理を行っている．後段では， NMS（Non Maximum Suppression）と呼ばれる，重複する検知結果を融合する処理を行なう．

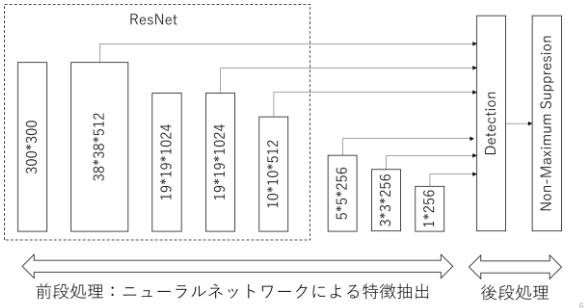

図 1 SSD-ResNet の構造

GPU における並列処理の効率化を検討する上で，推論処理の中で GPU のリソースがどのように使われているかを見ることは大切である．ここでは， Nsight プロファイラ[9]を利用し，可視化することにした（図 2）. 図 2 では，サーバー本体の CPU と GPU の処理負荷量と，CPU と GPU 間で転送されるデータ量を時間経過とともに見ることができる．SSD-ResNet では，最初にサーバー（CPU）から GPU に対して画像データの転送があり，その後，GPU 処理が集中する期間がある．この期間は，CPU はほとんど動いておらず, データ転送も行っていない. 次の後段の段階では，CPU の処理が増え，GPU の処理が減少し，データ転送が増えていることがわかる．これを図 1 と比較してみると，GPU の処理負荷が高い部分は，前段のニューラルネットワークの処理に該当すると考えられる．一方で，後段の検知処理と NMS 処理に相当する部分では，GPU の稼働している部分は少なく CPU での処理が中心となっている．また，短いながらも推論処理の前後には，推論に利用するデータを GPU のメモリに送る入力と，処理結果を GPU のメモリから取り出す出力も行っている．

このように，一口に推論処理といっても，その挙動は，データ入力，GPU 中心のニューラルネットワーク処理，CPU 中心のそれ以外の処理，結果出力と，GPU の利用傾向の異なる複数の要素で構成されており，推論処理中に常に GPU が動き続けていないことがわかる．このうち，ニューラルネットワークの処理部分が，いわゆる AI における推論処理の本体部分であり，ここが最も GPU を使う部分になる．

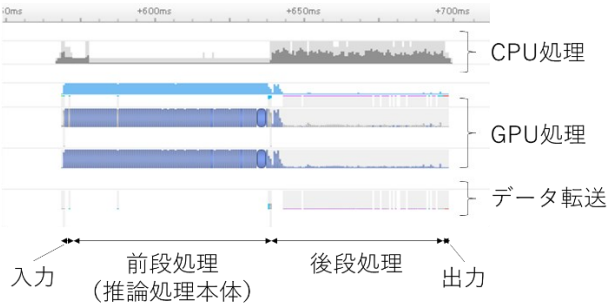

図 2 SSD-ResNet 処理時の GPU の挙動

以上から，推論処理を並列実行するには，GPU 処理が中心となるニューラルネットワークの処理期間において排他的に CUDA コアを利用できることと，サーバーと GPU との間でデータ転送が行われている期間の GPU を効率的に使うことが，GPU の利用効率化を図るためのポイントになると考えられる．

3. 提案手法

カメラ映像の画像解析に GPU を利用することがますます期待されるが，GPU が高価であることから利用効率を高めて相対的なコスト低減を実現することが望まれる．そのためには，(1) GPU に入力される処理量を適切にすること，（2）GPU 内の処理をできるだけ並列化して，GPU の空き時間を減らすこと，が重要である．図 3 に GPU 資源の利用効率化を実現するための処理概要を示した．

図 3 では，複数のカメラ映像に対して，複数の GPU で推論処理するケースを記載している．GPU の性能から，単位時間あたりに推論可能な量は決まっているため，カメラから送信される映像を適切なデータ量に変換してから GPU に入力する必要がある．ここでは，実行先制御部において，カメラから入力された映像を分割または統合することによって，GPU で処理可能なフレームレートに変換する. 実行タイミング制御部では，2 章で述べたように GPU での推論処理の段階を認識し，それに合わせて推論処理の実行タイミングを制御することによって，並列処理の高効率化を実現する部分である. 2 つの制御部はそれぞれ上記の (1) と（2）に対応する．

カメラ画像の処理においては，カメラから送られる映像は 30fps ないし 60fps である．しかし，受信したすべてのフレームを処理するには GPU に高い性能が必要となると同時に，画像解析でも必ずしも高頻度で推論処理を行う必要はない．例えば，交差点の監視システムでは，自動車の動きを認識する場合で 10fps 程度, 歩行者の場合には数 fps の

Vol.2021-DPS-186 No.65
Vol.2021-CSEC-92 No.65
2021/3/16

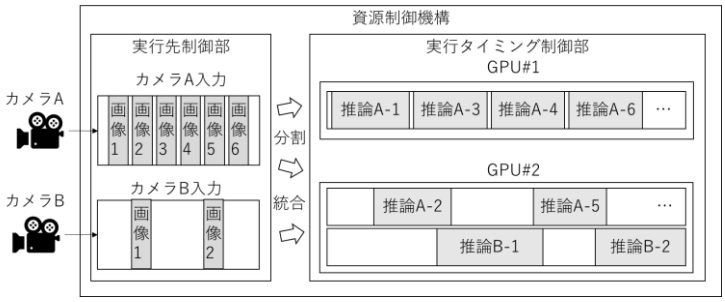

図 3　資源管理機構による推論処理の実行制御概念図

映像があれば良いと言われている．また，工場における品質検査や動きの少ない対象を監視するケースでは1fpsもしくはそれ以下というように，監視対象や目的に必要な最低のフレームレートは決まっている．従って，アプリケーションの要件に合わせて入力映像のフレームレートを落としたうえで，推論処理を行う．

GPU 内部での並列処理の効率化については，2.3 節で述べた推論処理中の推論処理の本体部分を利用して，推論処理の GPU への割り当てと実行タイミングを制御する．このために，提案手法では図 3 のように推論処理の GPU への割り当てを行う実行先制御部と，並列処理での推論処理の実行タイミングを制御する実行タイミング制御部からなる資源制御機構により，GPU の利用効率を向上する．

以降では，実行先制御部での推論処理の統合，分割と，実行タイミング制御部におけるタイミング制御の動作詳細，また，これを実現するシステム構成について説明する．

3.1 実行先制御

カメラ映像の入力のフレームレートに対して処理時間が短い場合，次のフレーム画像が推論処理可能になるまでの間に GPU を利用しない空きの時間が生じる．このようなカメラ映像処理が複数ある場合，推論処理を 1 つの GPU に統合して，空き時間中に別の推論処理を行うことで GPU の利用効率を向上させることができる．一方で，フレームレートに対して処理時間が長い場合は，1 つの GPU では処理が間に合わないため，2 つ以上の GPU に分割して処理する必要がある．分割した結果，推論処理の頻度が下がれば，これと別の映像処理とを統合するなどして，調整することも必要となる．

実行先制御部では，前述のように推論処理の頻度と，推論処理時間と本体部分の処理時間から，入力データを分割，統合を決定し対応する推論処理を GPU へ割り当てる．厳密には推論処理の GPU への割り当ては組み合わせ最適問題になるが，ここでは，最適でなくとも，すべての推論処理が GPU の処理性能内に収まるように実行先を決定する．

3.2 タイミング制御

複数の推論処理同士を統合して 1 つの GPU で並列処理を行う場合，統合した推論処理の処理時間を単純に合計しても GPU の処理時間に収まるなら，単純に推論処理の合間に別の推論処理を行うようにすればよい．一方で，推論処理時間の合計が GPU の処理能力を超えても，推論本体部分の合計が GPU の処理時間を超えなければ，適切に推論処理を重複させて並列に実行することで，処理することが可能である．

これまで述べてきたように，推論処理の本体部分は GPU の稼働率が高く，複数の本体部分を同時に実行してもスループットは向上せず，かえって双方の処理時間が増加してしまう結果になる．一方で，本体部分の処理中は GPU での処理が中心で，CPU での処理やデータ通信はほとんど行っていないため，入出力とは並列に実行しても互いの影響が少ないことが予想される．また，図 2 のように後段の処理部分でも GPU を利用しているが，ここでの GPU の稼働率は低いため，他の推論処理の本体部分や後段処理と重複実行しても MPS によるスループットの向上が期待できる．

つまりは，推論処理の本体部分同士を重複して実行しないようにさえすればよく，他の部分が重複しても処理時間の増加は少なく並列化によるスループットの向上が期待できる．そこで，実行タイミング制御部では，推論処理の本体部分が同時に実行されないように推論処理の実行タイミングを制御することで，推論処理の処理効率を向上させる．

3.3 システム構成

ここでは，複数台のカメラ映像に対して推論処理を行うシステムを想定する．推論処理はカメラ毎に独立したアプリケーションで行い，アプリケーションによっては異なる推論処理を行うこともある．また GPU を複数搭載しており，それぞれの GPU では MPS により複数の推論処理を並列に実行可能となっている．

各アプリケーションは AI フレームワークを利用して推論処理を行う．AI フレームワークには TensorFlow や TensorRT, PyTorch など様々な種類があり，アプリケーションに対して AI の学習と推論の機能を提供する．GPU を利用して推論を行う場合は，AI フレームワークから CUDA と GPU ドライバを介して GPU を利用することになる．

提案手法では，図 4 のように資源管理機構により AI フレームワークでの推論処理の実行を監視する．資源管理機構では推論処理の開始を検知すると．これに対して推論処理の実行先とタイミングを制御する。

Vol.2021-DPS-186 No.65
Vol.2021-CSEC-92 No.65
2021/3/16

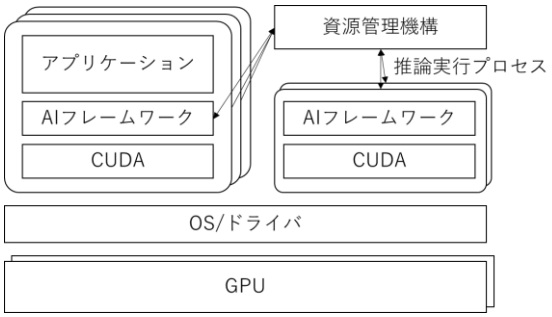

図 4　システム構成

この際に，検知した推論処理を分割しない場合はタイミングのみを制御して，同じ場所から推論処理を行わせる．一方で，分割が必要な場合は，推論実行プロセスに推論処理要求を送り，ここで推論処理を行わせる．分割数に応じて推論実行プロセスを用意し，複数の推論実行プロセスに対して推論処理要求を振り分けて送信することで複数 GPU に分割して推論処理することを可能にする．これにより，アプリケーションに手を加えずに様々な分割・統合のパターンに対応可能になる．

4. 評価

4.1 システム構成

資源管理機構には，推論の実行先を制御する機能と，推論の実行タイミングを制御する機能がある．ここでは，前述の SSD-ResNet を利用した物体検知を行うアプリケーションを複数実行する場合に対して，実行タイミング制御を行うことによるスループット向上の効果を評価する．

本評価では，GPU の性能に適したフレームレートの映像を入力することとして，並列処理の効率化のみを対象とする．そのため，1 台の GPU を利用し実行タイミング制御機能を中心に実現した．評価環境としては，AWS の GPU 搭載の仮想サーバーインスタンスを使用した．また，AI フレームワークには，広く普及している TensorFlow[10]を利用する．詳細なスペックを表 1 に記載する．

今回の評価では，デコードなどの映像処理の影響を排除するため，図 5 にあるように実際のカメラは使用せずに，あらかじめカメラで撮影した動画ファイルを仮想サーバー上に置いて利用した．また，検知結果は配信サーバーから映像ストリームとして出力し，リモートからブラウザ等で表示可能としている．1 台のインスタンス上でこのようなシステムを複数同時に稼働させ，資源管理機構で物体検知アプリケーションでの推論処理による GPU の利用を制御する．

表 1　評価システムスペック

インスタンス種別	g4dn.2xlarge
CPU	Xeon 8259@2.5 GHz x 8
メインメモリ	16 GB
GPU	NVIDIA T4
GPU メモリ	16 GB
ベースイメージ	Machine Learning AMI
OS	Ubuntu 18.04
CUDA バージョン	10.0.130
TensorfFow バージョン	2.3.0
GPU ドライババージョン	450.51.05
MPS	有効

一連のシステムを単体で動作させた場合, SSD-ResNet の推論処理にかかる時間はおよそ 70ms から 80ms であり，ファイルの入出力などアプリケーション側にかかる処理も含めると 10fps での処理が可能である．この処理を図 5 に示したシステムで実装し，処理タイミングの制御を行うことで並列処理を効率化して, 10fps 以上の推論処理が可能であることを確認する．

まず，図 2 で示した SSD-ResNet の動作から，並列処理におけるタイミング制御方法について検討する．SSD-ResNet における推論処理時間のおよそ 6 割をニューラルネットワーク処理，つまり推論処理の本体部分が占め，残りの 4 割を入出力や後段処理で占めている．Nsight の測定オーバーヘッドや処理のばらつきを加味しても，推論処理の本体部分は 50ms から 60ms であることが推測できる．GPU の推論処理で並列化できないのは，この推論処理本体部分であるため，仮にこの時間を 60ms とすると，1 秒間に 16 個の本体部分を処理可能な計算になる．この結果，タイミング制御を行ったうえで並列処理を行うと, 16fps のフレームレートで処理可能なことが期待できる．そこで，4fps の

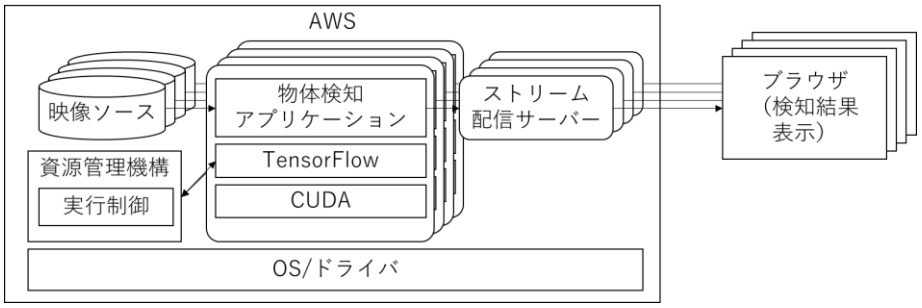

図 5　評価システム構成

推論処理を，実行タイミング制御により，4 並列で並列処理した場合に，全体で 16fps のスループットが達成可能なことを図 5 に示すシステムで確認する．

4.2 推論本体部分の調査

推論処理本体が同時に実行されないようにタイミング制御を行うためには，実際の環境で推論本体の処理時間を求める必要がある．そこで，図 5 のシステムを利用して 2 つの推論処理（SSD-ResNet）を同時に実行し，処理開始時間を少しずつ変えながら, 個々の推論処理時間を測定する（図 6）．2 つの推論処理における本体部分が同時に実行されれば, 個々の推論処理時間が長くなることが予想されるため，この結果から推論本体時間と後段の処理時間を見つけることにした．

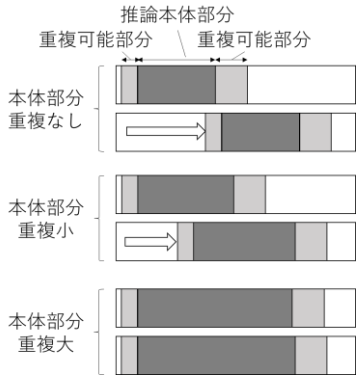

図 6　推論本体部分の重複と処理時間への影響

表 2 は開始時刻のずれを 0ms～100ms まで 5ms ずつ変化させたときの 2 つの推論処理の処理時間である．値は全て 50 回実行したときの平均値である．ただし，TensorFlow では初回の実行時に様々な初期化処理が発生し処理時間が増加するため，初回の処理時間を除いて集計している．表 2 で, 推論処理時間#1 が先行して実行した処理の処理時間で, 推論処理時間#2 がずらして実行した処理の推論処理時間になる．開始時刻差が 0ms，つまり完全に 2 つの推論処理を同時に実行した場合, 双方の処理時間が増加し, 約 125ms と単体で処理するよりも 1.5 倍の処理時間がかかる．推論処理時間は開始時刻差が広がるにつれて双方とも減少していく．また, 開始時刻差が 45ms を超えると，処理時間の減少幅が小さくなり，ほとんど同じ値になる．この時の，推論処理時間は 70ms から 80ms であり，これは SSD-ResNet を単体で実行したときとほぼ同じ時間となる．つまり，この 45ms が推論処理の本体部分の時間と推定できる．さらに，開始時刻差が 75ms を超えると，双方とも推論処理時間が単体実行のときと同じ時間とるため，推論処理全体が重複しなくなったと判断できる．

上記の結果から，SSD-ResNet を GPU で並列実行する際には，推論処理の開始タイミングを 45ms ずらして処理を行うことで，GPU を効率的に利用することができる．しかし，実際には処理時間には揺らぎがあるため，長時間にわたって安定動作させるには, 60ms 程度にすることが良いことがわかっている．資源管理機構にて推論処理を 60ms ずつずらして実行するようにすれば図 7 に示した模式図のように 4 つの推論処理を 250ms 間隔に収めることが可能だと考えられる．これにより 4fps の推論処理を 4 並列で動作可能となっている．

表 2　SSD-ReNet 重複実行結果

開始時刻差	推論処理時間#1	推論処理時間#2
0 ms	127.1 ms	125.2 ms
5 ms	123.7 ms	120.4 ms
10 ms	115.3 ms	114.4 ms
15 ms	107.3 ms	108.5 ms
20 ms	105.2 ms	105.5 ms
25 ms	98.4 ms	99.5 ms
30 ms	92.2 ms	92.7 ms
35 ms	91.8 ms	88.4 ms
40 ms	84.8 ms	84.3 ms
45 ms	76.7 ms	82.2 ms
50 ms	76.7 ms	82.2 ms
55 ms	78.4 ms	81.0 ms
60 ms	78.9 ms	82.7 ms
65 ms	78.6 ms	77.1 ms
70 ms	76.9 ms	75.3 ms
75 ms	72.2 ms	76.3 ms
80 ms	72.3 ms	73.8 ms
85 ms	71.6 ms	74.4 ms
90 ms	72.7 ms	73.0 ms
95 ms	70.8 ms	72.3 ms
100 ms	72.8 ms	73.3 ms

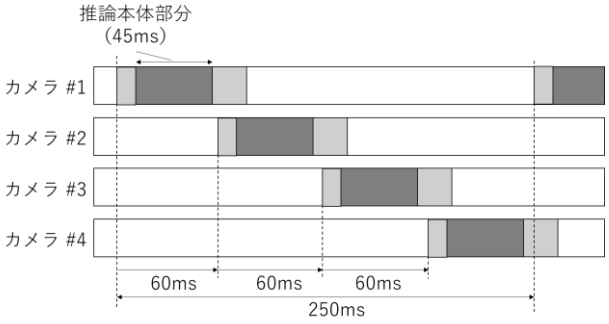

図 7　並列実行時の動作模式図

4.3 評価結果

図 5 に示した資源管理機構により 4 つのカメラ画像の物体検知において，推論処理の開始タイミングを制御し，その効果を評価する．それぞれのカメラ画像は任意のタイミ

Vol.2021-DPS-186 No.65
Vol.2021-CSEC-92 No.65
2021/3/16

ングで入力され，以降は 4fps，つまり 250ms の周期で到着し，推論処理される．このため，資源管理機構による制御機能が無い場合には，推論処理のタイミングはアプリケーションの起動タイミングにより依存し，推論本体部分の重なり方が変動するため，推論処理時間が一定しない可能性がある．

一方で，資源管理機構を有効にすると，カメラ入力のタイミングはばらばらであっても，TensorFlow に推論処理開始等のメッセージが通知されると，それを契機に資源管理機構が実行タイミングを制御して，4 つの推論処理が図 7 のように 60ms ずつずらして実行される．資源管理機構の効果を比較するために，機能を無効にして 4 つのカメラ画像入力が完全に同期する場合と，機能を有効にして 60ms ずれて処理される場合との測定結果を表 3 に示す．資源管理機構により推論処理の開始タイミングを制御した場合は，推論処理時間は平均で 83ms 程度と単体実行時よりも 1~2 割程度の増加となっており，最大でも 110ms と 250ms に収まるためコマ落は発生せず，4 つのアプリケーションが 4fps で動作するため，スループットは 16fps となる．

一方で，制御を行わない場合は，推論処理時間が平均で 268ms，最小でも 244ms と 250ms をわずかに超えない程度である．ここで，アプリケーションでは 250ms 周期で画像の入力が来るため，推論処理時間がこれを越える場合は，次フレームに対しての処理がスキップ，つまりコマ落ちしてしまう．このため，1 フレーム処理するごとにほとんど毎回コマ落ちが発生する．制御なしでのコマ落ち率は約 50%のため，スループットはコマ落ちにより半減して 8fps となり，ワーストケースとの比較ではフレームレートは 2 倍となる．SSD-ResNet 単体では 10fps 程度で動作するため，単体と比較しても 1.6 倍に向上させることが可能となる．

表 3　評価結果

	制御あり	制御なし
最小値	77 ms	244 ms
最大値	110 ms	303 ms
平均値	83 ms	268 ms
コマ落ち率	0 %	50%
スループット	16 fps	8 fps

並列実行同士の比較では，制御を行わない場合アプリケーションの起動のタイミングにより処理時間が大きくばらつくため，定量的な性能の比較は容易ではないが，ワーストケースを想定するとフレームレートか同時実行数を落とす必要があり，16fps のスループットは達成不可能なことになる．一方で，本手法では資源管理機構による性能のばらつきを抑えることで性能マージンを削減し，16fps のスループットが達成可能となる．

5. おわりに

近年，GPU を利用した推論処理が増加しているが，さらなる普及のためには複数の推論処理で高価な GPU を効率的に共有することで，処理コストを削減する必要がある．

本稿で提案した手法では GPU に入力する処理量をアプリケーションの要件に基づいて適切に制御することと，GPU 内の処理を並列化して処理の空き時間を最小にすることで，高価な GPU を複数のアプリケーションから効率的に利用することを可能にし，推論処理の低コスト化を実現する．

今回は，このうちの並列処理での GPU 利用の効率化について，推論処理の実行タイミングを制御することで，従来技術では効率化が難しい推論処理本体部分の重複実行を回避しつつ，GPU の空きを別の推論処理で埋めてスループットを向上する技術について評価を行い，SSD-ResNet に対してワーストケースと比較して 2 倍，単体実行との比較でも 1.6 倍のスループットの向上が可能なことが確認できた．

しかしながら，本手法で必要となる推論処理の本体部分を探し出すために，現状では事前に推論処理を実行して調査を行い，本体部分を回避するための動きをあらかじめ決めておく必要がある．また GPU への割り当てについてはまだ手動で行っており，これらが今後の課題となる．

参考文献

[1] Kaiming He, Xiangyu Zhang, Shaoqing Ren, Jian Sun.. Deep Residual Learning for Image Recognition. Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR), 2016, pp. 770-778
[2] Joseph Redmon, Santosh Divvala, Ross Girshick, Ali Farhadi.. You Only Look Once: Unified, Real-Time Object Detection. Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR), 2016, pp. 779-788
[3] “バッチ予測の取得”. https://cloud.google.com/ai-platform/prediction/docs/batch-predict?hl=ja,(参照 2021-02-05)
[4] “Optimizing Performance with TensorRT 2.2”. https://docs.nvidia.com/deeplearning/tensorrt/best-practices/index.html#batching,(参照 2021-02-05)
[5] “CUDA Toolkit Documentation”, https://docs.nvidia.com/cuda/,(参照 2021-02-05)
[6] “Multi Process Service”, https://docs.nvidia.com/deploy/pdf/CUDA_Multi_Process_Service_Overview.pdf,(参照 2021-02-05)
[7] “CUDA Toolkit Documentation 2.1 Kernels”.. https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#kernels(参照 2021-02-05)
[8] X. Lu, X. Kang, S. Nishide and F. Ren.. Object detection based on SSD-ResNet. 2019 IEEE 6th International Conference on Cloud Computing and Intelligence Systems (CCIS), Singapore, 2019, pp. 89-92, doi: 10.1109/CCIS48116.2019.9073753.
[9] “NVIDIA Nsight Systems”. https://developer.nvidia.com/nsight-systems(参照 2021-02-05)
[10] Martin Abadi, et al.. TensorFlow: A System for Large-Scale Machine Learning. 12th Symposium on Operating Systems Design and Implementation (OSDI) 16. 2016, pp265-283.