Vol.2019-HPC-170 No.14
2019/7/24

SX-Aurora TSUBASA の Deep Learning への適用

大野 善之[1,a)]　井手口 裕太[1]　石坂 一久[1]

概要：SX-Aurora TSUBASA は，汎用の Xeon サーバに，ベクトルプロセッサを搭載した PCI カード (Vector Engine, VE) を接続した，新しいコンピューティングシステムである．従来，ベクトルプロセッサが利用されてきた科学技術計算だけでなく，幅広い領域での活用が期待されている．本研究では，SX-Aurora TSUBASA の新しい活用先として，Deep Learning(DL) で活用することを目標とし，既存の DL フレームワークからアクセラレータとして VE を利用する機構について報告する．我々は，DL フレームワークとして TensorFlow を選択し，TensorFlow の GPU 拡張を参考にして，アクセラレータとして VE を利用する拡張を実装した．また，既存の実装では DL の処理を構成する各演算単位でデバイスにオフロード行うため，比較的小規模な DL モデルでは，デバイス側での演算よりもオフロードのコストが支配的になり，性能低下を引き起こすことに着目した．本稿では，個々の演算単位ではなく学習全体をオフロードする Macro Offload を提案する．TensorFlow に，学習ループ全体を実行するような新しい operation を追加して，疑似的に Macro Offload された状態を作ったところ，オフロードのコストなく実行できることを確認した．

キーワード：SX-Aurora TSUBASA, Deep Learning, TensorFlow

1. はじめに

近年，脳の神経回路網を模倣した Neural Network を多層に組み合わせた機械学習手法である Deep Learning が注目されている．Deep Learning により，認識や検出，判断などを高精度に行うことが可能になっている．高精度に認識や検出ができる一方で，Deep Learning を実行するにあたっては，高い計算機性能（演算性能，メモリ性能）が必要とされる．

SX-Aurora TSUBASA[1] は，Vector Host(VH) と呼ばれる x86 サーバに，PCI-Express 経由で Vector Engine(VE) と呼ばれるベクトルプロセッサが搭載されたカードを接続した構成をとるコンピューティングシステムである．VE は単精度演算性能 4.3 TFlops，メモリバンド幅 1.2 GB/s という，高い計算性能を有する．また，従来のベクトルプロセッサと異なり，汎用の Linux 環境で動作するため，従来ベクトルプロセッサが活用されてきた科学技術計算領域だけではなく，幅広い用途での活用が期待されている．

しかしながら，Deep Learning に使われる Convolution 演算などの基本演算を VE で動作させたという報告 [2] はあるものの，SX-Aurora TSUBASA において Deep Learning を動作させたという報告はまだない． 現在，Deep Learning を動かすには，TensorFlow[3] や PyTorch[4]，Chainer[5] といった Deep Learning フレームワークを用いて動かすことが一般的である．SX-Aurora TSUBASA で Deep Learning を動かすためにも，Deep Learning フレームワークから VE が利用できるようになる必要がある．そこで，筆者らは，世の中で広く利用されている Deep Learning フレームワークの 1 つである TensorFlow から，VE に演算をオフロードして動かすための拡張実装を行った．本稿の前半では，筆者らが行った TensorFlow の VE 拡張実装について述べる．

筆者らの実装では，GPGPU と同じように VE をアクセラレータとして用い，個々の演算をオフロードするという実行モデルを採用した．このようなオフロードモデルでは，Deep Learning モデルが小さく，個々の演算の実行時間が小さい場合はデバイスを利用するコストが問題となることがわかった．Narayanan ら [6] も，Deep Learning において，オフロードのオーバーヘッドのためアクセラレータの性能を活かせない点を問題視している．Narayanan らは，複数の小規模な Deep Learning モデルを並列に GPU で動作させるというワークロードに対して，複数のモデル間にまたがる演算や通信を融合させ，アクセラレータの使用率を向上させる方式を提案している．我々は，本問題に対する 1 つの解決策として，Deep Learning の個々の演算をオフロードするのではなく，学習ループ全体をオフロー

[1] NEC データサイエンス研究所
a) y-ohno@ji.jp.nec.com

© 2019 Information Processing Society of Japan

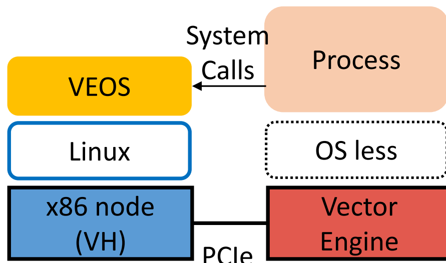

図 1　SX-Aurora TSUBASA のシステムアーキテクチャ

ドする Macro Offlaod モデルについて提唱する．本稿の後半では，Macro Offloading モデルについて，実行モデルの概要，および，効果を確かめるための簡易評価について報告する．

本稿の構成は以下のとおりである．2 章で SX-Aurora TSUBASA のシステムアーキテクチャ，実行モデルについて説明し，3 章で TensorFlow について簡単に説明する．4 章で TensorFlow の VE 拡張について，前半で基本構造を説明し，後半で高速化のための仕組みについて述べ，5 章では，提案する Macro Offloading モデルの概要とその効果を効果を確かめるための簡易評価について述べる．そして，最後の 6 章でまとめを述べる．

2.　SX-Aurora TSUBASA

SX-Aurora TSUBASA[1] は，Vector Host(VH) と呼ばれる x86 サーバに，PCI-Express 経由で Vector Engine(VE) と呼ばれるベクトルプロセッサが搭載されたカードを接続した構成をとるコンピューティングシステムである．

図 1 に SX-Aurora TSUBASA のシステムアーキテクチャを示す．VH は，標準的な x86 の Linux サーバである．したがって，x86 向けのアプリケーションは VH でそのまま動作させることができる．一方，VE は Linux OS から見ると，1 つの PCI デバイスにすぎない．VEOS という VE の OS 相当の機能を提供するソフトウェアが VE の制御を行い，VE にプロセスを生成したり，VE が発行するシステムコールを肩代わりすることで，VE でプログラムを実行させることができる．

2.1　SX-Aurora TSUBASA の実行モデル

2.1.1　Native モデル

SX-Aurora TSUBASA の最も基本的な実行モデルは，VE カード 1 枚を 1 つのスタンドアローンなコンピュータとみなし，プログラム全体を VE で実行させるモデル (Native モデル, (図 2 上段)) である．

Native モデルは，VE 向けの実行バイナリを実行するだけで，VE 上でプログラム全体が動作するという実行形態である． 実際には すべてが VE で動いているわけではなく，VE 上のメモリやプロセスの管理，VE 上に立ち上げたプロセスが発行したシステムコールの処理等は，VH で動作している VEOS がその役割を担っている．しかしながら，ユーザは VEOS の存在を意識してプログラムを実行させる必要は特になく，C，C++ および Fortran で書かれたプログラムを VE 向けコンパイラでコンパイルして，実行するだけでプログラムを動作させることができる．

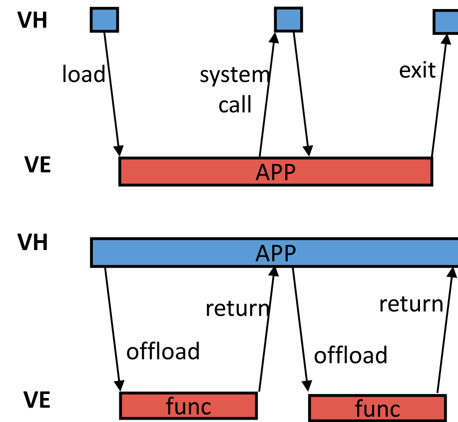

図 2　Native モデル (上段) と Offload モデル (下段)

2.1.2　Offload モデル

一方，GPGPU のように VE をアクセラレータとして用いる実行モデル (Offload モデル, (図 2 下段)) も可能である．Offload モデルで実行するには，Vector Engine Offloading(VEO)[7] というフレームワークを用いる．VE にオフロードさせたい関数を VE 向けコンパイラでコンパイルしてライブラリ化しておき，VH 側のメインプログラム内で，VEO の API を利用して VE 側の関数を呼び出すことで VE に処理をオフロードさせることができる．

以下，オフロードに用いる VEO の API の一部を紹介する．

`veo_args_set_stack`
: VE で実行する関数に与える引数をスタックにセットする

`veo_call_async`
: セットした引数とともに VE の関数を非同期に呼び出す

`veo_call_wait_result`
: 呼び出した VE の関数の実行を待つ

これらを用い，呼び出したい関数に与える引数を設定し，関数を呼び出し，結果を待つというフローで VE へ処理のオフロードを行う．また，VEO では，関数のオフロード用の API だけでなく，VH から VE 側のメモリ確保を行う API(`veo_alloc_mem`，`veo_free_mem`) や，VH から VE に対してデータ転送をしかける API(`veo_read_mem`，`veo_write_mem`) も提供されており，VE をアクセラレータとして用いる実行モデルをサポートしている．

2.1.3　実行モデルの比較

前節までで，VE にプログラムを実行させる Native モデルと Offload モデルの 2 つの実行モデルを簡単に説明し

© 2019 Information Processing Society of Japan

た．表 1 に，それぞれの特徴をまとめる．

Native モデルでは，アプリケーションプログラム全体を VE 向けのコンパイラでコンパイルするだけで実行することができるため，容易にプログラムを実行させることができる．一方，Offload モデルでは，アプリケーションプログラムを VH で動作させる部分と，VE で動作させる部分を切り分け，連携させて動かすようにプログラムしなければならない．また，Offload モデルでは，VE への処理のオフロードの粒度が小さいと Offload のオーバーヘッドが問題になる可能性もある．逆に，Native モデルの欠点もある．プログラム全体を VE で動作させることになるため，VE で不向きな処理まで VE で動作させ性能ボトルネックとなる可能性がある．VE の不向きな処理の例として，I/O などのシステムコールが支配的な処理，SIMD 化されていないプログラムなどベクトルプロセッサで動かしても速くならない処理が挙げられる．一方で，Offload モデルは，はじめから VE で動作させるべき処理を切り出して，明示的に VE で動作させるのでこのような問題はない．

このように，Native モデルと Offload モデルとで利点・欠点があるので，アプリケーション開発においては，どの実行モデルを採用するかは適切に設計する必要がある．

TensorFlow を含め，PyTorch[4]，Chainer[5] といった Deep Learning フレームワークの多くは，バックエンドは C/C++ でプログラムされているが，ユーザプログラムは Python で記述される．SX-Aurora TSUBASA でこのようなフレームワークを動作させるにあたり，Native モデルか Offload モデルのどちらを採用するかを考える．Native モデルの場合，全てのソフトウェアスタックを VE 用に移植しないとアプリケーション全体を動作させることができない．ユーザが Python のモジュールを利用したい場合，それが VE に移植されていないと，そのモジュールがないがために他の全ても動かせなくなることになる．Native モデルで VE 向けにアプリケーションを移植するには VE 用コンパイラでコンパイルすれば動作させることができるが，SIMD 化されていないプログラムなどベクトルプロセッサにむかない処理を実行すると遅くなる可能性がある．以上を踏まえて，本稿では，Offload モデルで VE を用いて Deep Learning フレームワークを動かす方針を採用する．

3. TensorFlow

TensorFlow[3] は，Google が開発しオープンソースで公開している機械学習用のフレームワークである．CPU だけでなく，GPGPU や TPU といった異種の計算デバイスを用いて演算を行うことができる．

以下，TensorFlow の実行の流れ，特に異種の計算デバイスにまたがって実行される場合について説明する．TensorFlow の計算は，有向グラフとして表現される．グラフの個々のノードは，operation と呼ばれる個々の演算処理に対応し，エッジは Tensor と呼ばれる多次元配列データの流れを表している．ユーザは，TensorFlow で提供される C++ や Python の API を使い，計算グラフの定義や実行を行う．図 3 は，行列積 $A \times B + C$ だけを行う単純な Python プログラムとその計算グラフの例である．プログラムの前半では，定数行列 A，B，C および，それらの行列を用いた演算 $A \times B + C$ を定義している．sess.run() が呼ばれたとき，TensorFlow 内部では，図 3 の下のような計算グラフを構築し，計算グラフを評価した結果がユーザに返る．

```
A = tf.constant([1,2,3,4,5,6], shape=[2,3])
B = tf.constant([7,8,9], shape=[3,1])
C = tf.constant([10,11], shape=[2,1])
o = tf.matmul(A,B) + C

with tf.Session() as sess :
  result = sess.run(o)
```

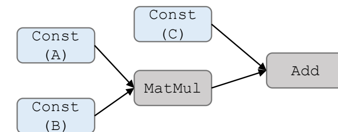

図 3　TensorFlow のプログラム例と計算グラフ

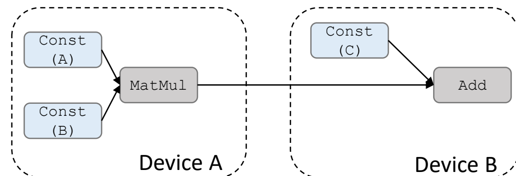

図 4　計算グラフのデバイス割り当て

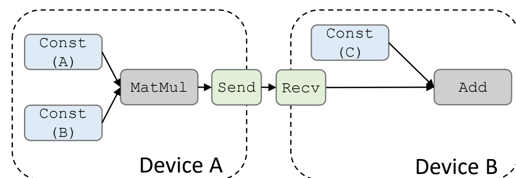

図 5　計算グラフへの転送 operation の追加

異種デバイスが用いられる場合は，少し処理が複雑になる．計算グラフを構築したあとに，グラフの各 operation をどのデバイスで実行させるかどうかの割り当てを決定する．決定には，operation にデバイス用の実装 (kernel と呼ばれる) が実装されているか，同一デバイスに割り当てるべき operation に関する制約，デバイスの優先度といった情報を考慮して決定される．例えば，図 3 の計算グラフが図 4 のように 2 つのデバイスに分割して割り当てられる．複数のデバイスに計算グラフが分割されると，デバイス間で Tensor のやり取りの必要が生じるため，最終的に，図 5 のように， デバイス間で Tensor の転送を行う operation (Send/Recv) を追加した計算グラフが作成され，演算が実行される．

TensorFlow に新規のデバイスを追加する場合は，デバ

表 1　Vector Engine の実行モデルの特徴

Native モデル	Offload モデル
○ 実行が容易（コンパイルして実行するだけでよい）	▲ オフロード用にプログラムの修正が必要
	▲ オフロードに伴うオーバーヘッドがある
▲ ベクトル向きでない処理まで実行	○ ベクトル向きの処理のみ実行

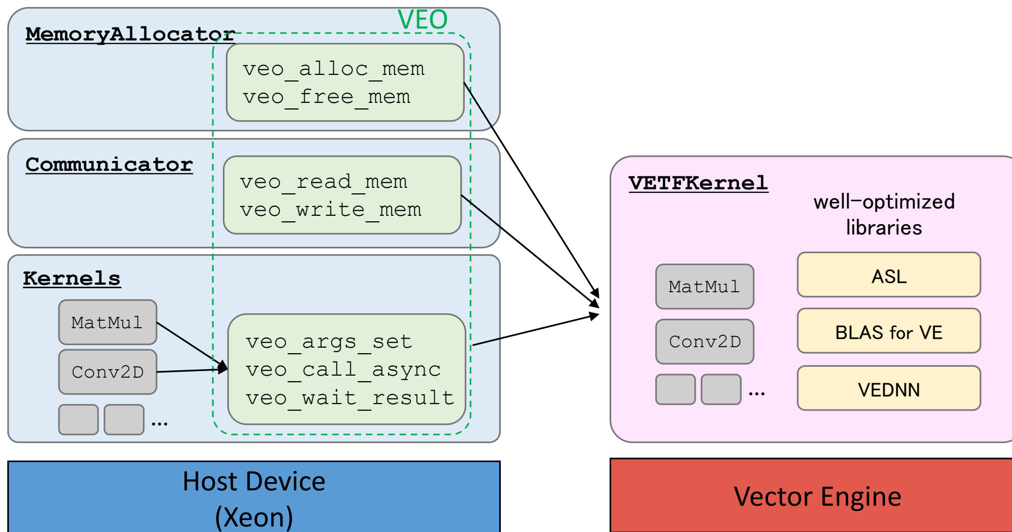

図 6　TensorFlow の VE 拡張の概要

イスのメモリアロケータ機能，デバイスとホストデバイス (CPU) との間のデータ転送機能，各 operation のデバイス用の kernel 実装の 3 点を追加をしておけば，実行時にそのデバイスがある場合は，先述の異種デバイスを用いる場合の実行フローによりデバイスを使って実行が行われる．

4.　TensorFlow の VE 拡張

4.1　VEO を用いた基本実装

図 6 に，VEO を用いた TensorFlow の VE 拡張の概要を図示する．2 章で説明したとおり，VEO では，VE 側のメモリアロケータ機能，VE と VH のメモリ転送機能，VH から VE 側の関数を実行する機能がある．我々は，VEO を利用し，新規デバイスの追加に必要な 3 機能（デバイスのメモリアロケータ機能，デバイスとホストデバイスとの間のデータ転送機能，各 operation のデバイス用の kernel 実装) を実装した．

デバイスのメモリアロケータ機能，デバイスとホストデバイスとの間のデータ転送機能については，ホスト側で VEO の API をそのまま実行することで，デバイス上のメモリの確保やホストデバイス間の通信を行っている．

個々の kernel については，実際の演算を行う VE 側の実装と，VE 側の関数を VEO を利用して呼び出すホスト側の実装とを追加した．VE 側の Kernel 実装は，VEO で VH から呼ばれるインターフェースとなる関数 を Kernel ごとに用意し，実際の演算部分に関しては，各 Kernel の演算に応じて，VE の演算性能を引き出せるような実装を採用した．行列計算や乱数生成などは NEC が提供する科学技術計算用のライブラリ群の中の VE 用 BLAS や ASL を利用した．また，Convolution や MaxPooling，Activation といった各 Deep Learning Layer については，筆者らが開発している VE 用 DNN Library (VEDNN[*1]) を利用している．また，Tensor の各要素を Element-Wise に計算するようなプリミティブな演算についてはアセンブリレベルで実装をしている．

4.2　オフロード回数の削減

前節のように，VEO の機能をそのまま使うことで，TensorFlow の VE 拡張の実装を実現することができる．本節では，さらに前節の基本実装をベースに高速化を図った点について述べる．

各カーネルごとに VEO を用いてオフロードを行っていた場合，各カーネルの実行時間が短い場合，オフロードにかかるオーバーヘッドが問題になる可能性がある．そこで，各カーネルごとにオフロードするのではなく，部分グラフをまとめてオフロードする機能を実装した．

図 7 に，複数カーネルをまとめてオフロードする仕組みを図示する．個々のカーネルを VEO でオフロードする場合は，カーネルの VE 側関数に与える引数をセットして，VEO で VE 側関数を呼び出すというフローであった．複数カーネルをまとめてオフロードする場合は，カーネルの VE 側関数に与える引数，および，呼び出す VE 側関数のアドレスを，順次 queue に登録していき，queue 全体を引数として VEO で VE 側関数（`vetfkl_entry`）を呼び出す．

[*1] https://github.com/sx-aurora-dev/vednn で OSS 公開

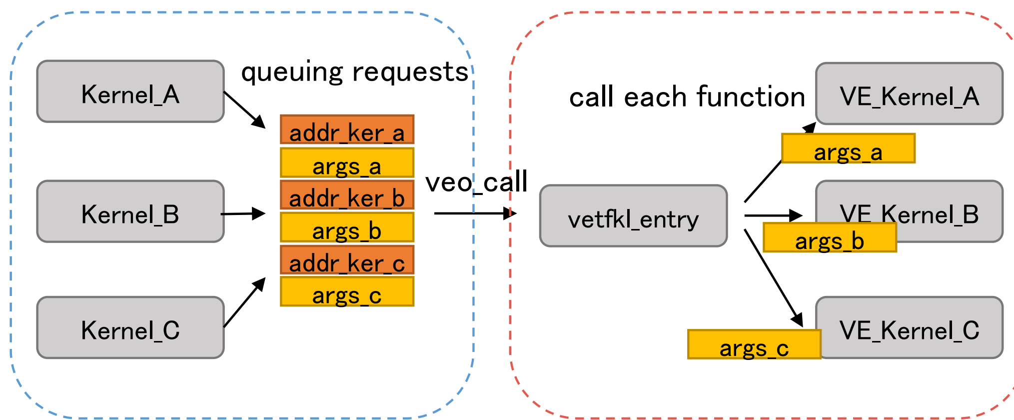

図 7　複数カーネルをまとめたオフロード

`vetfkl_entry` では，引数の queue から VE 側関数のアドレスとその引数を取り出し，順次関数を実行する．このような仕組みで，VEO のオフロード回数の削減を行った．

5. Macro Offload モデル

5.1 Offload モデルの問題点

前章では，VEO を用いて，TensorFlow の演算を VE にオフロードする基本実装，および，複数カーネルをまとめて VE にオフロードする機能について述べた．前章でも述べたとおり，複数カーネルをまとめてオフロードするようにした理由は，各カーネルの実行時間が短い場合，オフロードにかかるオーバーヘッドが問題になる可能性があるためである．

マルチチャネル，マルチバッチサイズの convolution など，1 つのカーネルの実行時間が長い場合は問題にならないが，個々のカーネルの処理量が小さくなるような Deep Learning モデルでは，問題になる．

例えば，図 8 に，VE および NVIDIA Tesla V100 で，TensorFlow を用いて比較的小規模なモデルで学習を行った場合のタイムラインを示す．使用したモデルは，1024 次元のベクトル値を入力として，2 値に分類する 全結合層 3 層のモデルである．タイムラインは，上段が VE の結果，下段が V100 の結果である．それぞれのタイムラインは 3 段あり，上段がデバイス側で演算を行っている時間，中段がホストデバイス (CPU) で実行される kernel の実行時間，下段がデバイス側で実行される kernel を呼ぶためにホスト側の処理を行っている時間である．VE も V100 も，デバイス側が動いている時間が短いことがわかる．なお，デバイスの実行時間が，V100 は分散しているが，VE では 1 箇所に集まっているのは，VE 側では 複数カーネルをまとめて VE にオフロードしているためである．このように，個々のカーネルの処理量が小さい場合，オフロードにかかるオーバーヘッドが支配的になる．

5.2 Macro Offload モデル

個々のカーネルの処理量が小さいような Deep Learning モデルで，オフロードのオーバーヘッドのためにアクセラレータの性能を活かせない問題に対して，新しいオフロードモデルである Macro Offload モデルを提案する．

以下に，Deep Learning の学習スクリプトの基本フローを示す．

```
1  データセットのロード
2  モデルの生成
3  loop(epoch) {
4    loop(dataset) {
5      ミニバッチの選択
6      選択したミニバッチを用いてモデル更新
7    }
8    テストデータで精度のテスト
9  }
```

基本的な Deep Learning の学習スクリプトは，学習に用いるデータセットの読み込みを行い，モデルを生成し，ループを回してモデルの更新を続けるというフローからなる．モデル更新のループは，全データをミニバッチに分割し学習を行うというループ (4-7 行目，ミニバッチループ) と，その繰り返しを何セットも繰り返すというループ (3-9 行目，エポックループ) の多重ループとなる．

TensorFlow では，上記のエポックループ，ミニバッチループをユーザが明示的に書き，ミニバッチを用いたモデル更新のたびに sess.run() を行うという実行フローが一般的である．デバイスを用いるときには，sess.run() が呼ばれると，各 kernel ごとにオフロードが行われている．このような 個々の演算単位でオフロードする (図 9 左) という Kernel Offload に対して，我々は，上記のループ構造に限定することで，ミニバッチループ全体，または，エポックループとミニバッチループの多重ループ全体をまとめて，比較的大きい粒度でデバイスにオフロードをする（図 9 右）MacroOffload を提案する．

Macro Offload モデルを，既存の Deep Learning フレームワークで適用しようとすると，フレームワーク全体の

© 2019 Information Processing Society of Japan

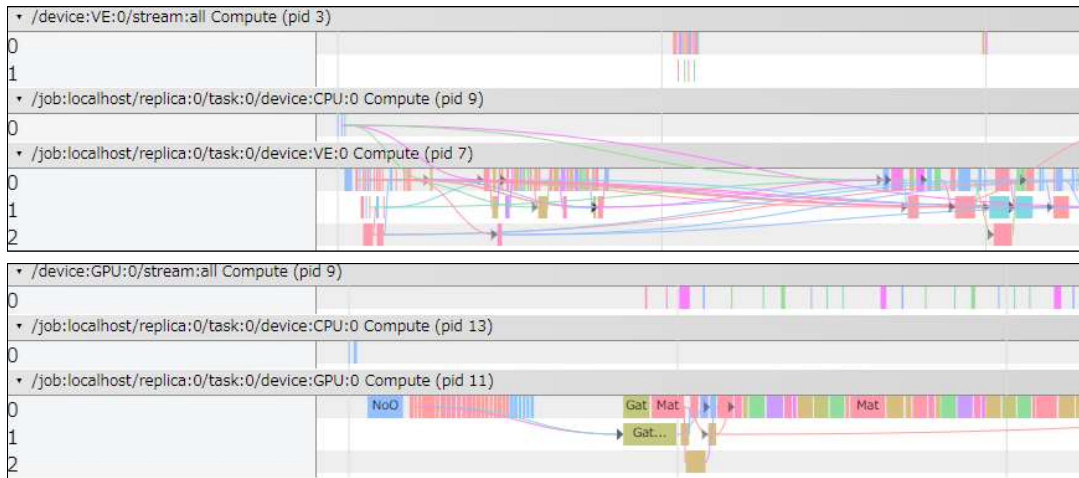

図 8 モデルが小さい場合の timeline (上段：VE，下段:V100)

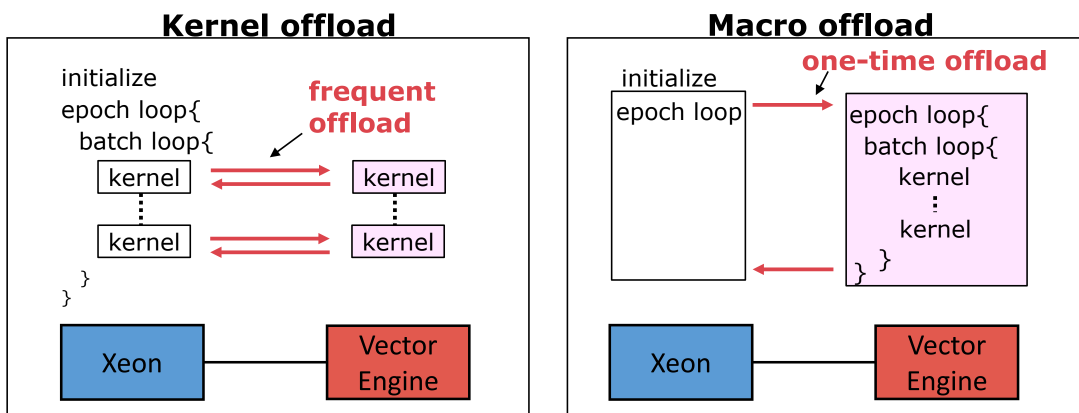

図 9 Kernel Offload と Macro Offload

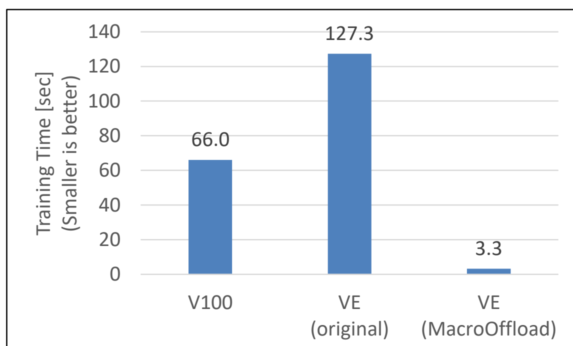

図 10 MacroOffloading の効果（学習時間）

設計から見直す必要がある．本設計に関しては，検討中である．我々は，Macro Offload の効果を確認するため，TensorFlow に，モデル定義やデータセットを入力とし，エポックループとバッチループの多重ループを実行するような，Macro Offload 後の状態を模擬する新しい operation を作成した．

図 10 に，上述の Macro Offload を模擬する operation を用い，5.1 節で用いた小規模な全結合 3 層のモデルと，20 万個の 1024 次元のデータセットを用いて実行した場合の 1 epoch の学習時間を示す．通常の VE 版 TensorFlow で実行した場合 127.3sec かかっていたところ，Macro Offload では 3.3sec で終わっており，細かなオフロードが性能低下を引き起こしていたことがわかる．

6. おわりに

本稿では，SX-Aurora TSUBASA を Deep Learning に活用することを目標に，世の中で広く利用されている Deep Learning フレームワークの 1 つである TensorFlow を SX-Aurora TSUBASA を用いて動かすために実施した拡張について報告した．本稿の TensorFlow の VE 拡張では，VEO を用い，GPGPU 版のように，各 kernel をデバイスにオフロードするという実装を採用し，VE の活用を実現した．

しかしながら，各 kernel をデバイスにオフロードする方式では，個々の kernel の演算時間が短くなるような Deep Learning モデルでは，アクセラレータの高い性能を活かしきれないという問題がある．そこで，本稿では，Macro Offload と呼ぶ，学習全体をデバイスにオフロードするモデルについて提案した．TensorFlow に対して，エポックループとバッチループを回すような operation を実装して，擬似的に Macro Offload した場合を作り出し，オフロード

© 2019 Information Processing Society of Japan

にかかるオーバーヘッドを削減できることを確認した．

今後，TensorFLow のような既存の Deep Learning フレームワークにおいて，どのように Macro Offload を実現させせるかという課題に取り組んでいく．

参考文献

[1] Yamada, Y. and Momose, S.: Vector Engine Processor of NEC's Brand-New Supercomputer SX-Aurora TSUBASA, *Proceedings of the 30th Symposium on High Performance Chips*, HC30 (2018).

[2] Kruus, E.: Deep Learning operation On SX-Aurora, SC18 NEC Booth Presentation, Available at https://www.hpc.nec/api/v1/forum/file/download?id=LE6ha4.

[3] Abadi, M., Agarwal, A., Barham, P., Brevdo, E., Chen, Z., Citro, C., Corrado, G., Davis, A., Dean, J., Devin, M., Ghemawat, S., Goodfellow, I., Harp, A., Irving, G., Isard, M., Jia, Y., Jozefowicz, R., Kaiser, L., Kudlur, M., Levenberg, J., Mane, D., Monga, R., Moore, S., Murray, D., Olah, C., Schuster, M., Shlens, J., Steiner, B., Sutskever, I., Talwar, K., Tucker, P., Vanhoucke, V., Vasudevan, V., Viegas, F., Vinyals, O., Warden, P., Wattenberg, M., Wicke, M., Yu, Y. and Zheng, X.: TensorFlow: Large-Scale Machine Learning on Heterogeneous Distributed Systems (2015).

[4] Paszke, A., Gross, S., Chintala, S., Chanan, G., Yang, E., DeVito, Z., Lin, Z., Desmaison, A., Antiga, L. and Lerer, A.: Automatic differentiation in PyTorch, *NIPS-W* (2017).

[5] Tokui, S., Oono, K., Hido, S. and Clayton, J.: Chainer: a Next-Generation Open Source Framework for Deep Learning, *Proceedings of Workshop on Machine Learning Systems (LearningSys) in The Twenty-ninth Annual Conference on Neural Information Processing Systems (NIPS)* (2015).

[6] Narayanan, D., Santhanam, K., Phanishayee, A. and Zaharia, M.: Accelerating Deep Learning Workloads Through Efficient Multi-Model Execution, *Workshop on Systems for Machine Learning(MLSys)* (2018).

[7] NEC: SX-Aurora TSUBASA Vector Engine Offloading (VEO), https://github.com/veos-sxarr-NEC/veoffload.

© 2019 Information Processing Society of Japan