Vol.2019-ARC-240 No.39
Vol.2019-SLDM-191 No.39
Vol.2019-EMB-53 No.39
2019/2/28

Simulinkモデルから CPU とアクセラレータの併用コードの作成手法

甲斐　琢朗[1,a)]　森　裕司[2]　枝廣　正人[1]

概要：Simulink モデルはブロック図であるため，モデルレベルでの高速化手法としてこれまでタスク並列化やパイプライン実行が知られていたが，画像処理や知能処理等のデータ並列処理はモデルのグラフ解析では高速化されなかった．本研究では Simulink モデルからデータ並列化を行うための手法を提案する．今回はデータ並列を行う環境として OpenVX と OpenCL を採用した．提案手法を用いて評価を行ったところ，最大 117 倍ほどの高速化を行うことに成功した．

Code generation from Simulink model on both CPU and accelerator

1. はじめに

近年車載制御においてその制御はモデルを用いて設計することが多く，特に日本では MathWorks 社の MATLAB[1] のツールの一つである Simulink[2] を用いて設計を行うことが多い．同時に，プロセッサ自体の性能向上が頭打ちになってきていることから車載制御の分野でもメニーコア・マルチコアでの処理を行うことが多くなってきた．これらのことから，Simulink モデルから並列コードを作成するということに対する需要が高まりつつある．

しかし，自動運転の時代になり，車載制御でも画像処理や知能処理など，データ並列的な処理をアクセラレータを用いて実行されることが増えてきた．

そのため，Simulink モデルからデータ並列性のあるプログラムを作る手法についての研究が求められている．しかしそれには Simulink でどのようにデータ並列性のあるモデルを記述するのか，どのようにそれを CPU と GPU(Graphics Processing Unit) や DFP(データフロープロセッサ [3]) のようなアクセラレータを併用して実行できるコードに変換するのかという課題がある．

本研究ではデータ並列に向けた Simulink モデルの作成方法，および Simulink モデルからデータ並列を行うアクセラレータ併用環境に向けたコードの作成方法についてある程度の指針を作成した．同時にこのような環境に向いていないモデルの特徴についてもまとめた．

2. データ並列向け Simulink モデル作成

この章ではデータ並列向け Simulink モデルに求められる条件とそれを満たすモデルの作成方法について述べる．

2.1 条件

今回は Simulink モデルからのアクセラレータ併用環境向けデータ並列コードを図 1 に示した方法で作成する．そのため，Simulink モデルは以下の項目を満たしていることが望ましい．

- アクセラレータ向けコードがそうであるように，モデルも SIMD(Single Instruction, Multiple Data streams) であることが望ましい
- アクセラレータ向けコードとして切り出しやすくするために，データ並列箇所が区別できる構造が望ましい
- アクセラレータ向けコードで必要となる，並列用の ID を設定しやすい
- タスク向け並列性も同時に行いたいため，タスク並列性を抽出しやすい構造である DAG(非巡回有向グラフ) となっている

2.2 モデルの作成手法

Simulink モデル上でデータ並列を行えるブロックにはい

[1] 名古屋大学
Nagoya University
[2] 株式会社エヌエスアイテクス
NSITEXE,Inc.
[a)] takuro_k@ertl.jp

© 2020 Information Processing Society of Japan

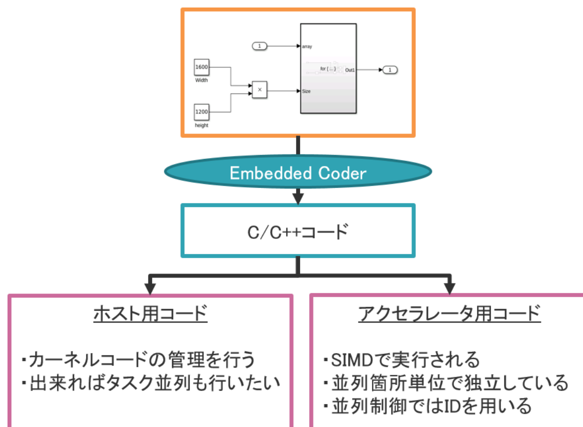

図 1 Simulink モデルからのデータ並列コード変換フローチャート

くつかの種類がある．それらの違いを表 1 にまとめた．

これらを比較した結果，本研究では並列箇所の区別が可能で，なおかつ ID の設定も容易な ForIterator サブシステムを採用することにした．

3. 使用言語の決定

この章ではデータ並列を行う様々な言語のうち，採用した言語の形式について説明する．

アクセラレータ向けデータ並列言語としては CUDA[4] や OpenVX[5]，OpenCL[6] が挙げられる．これらの違いを表 2 にまとめた．

今回はグラフ構造との相性を重視して OpenVX を採用した．しかし，OpenVX だけでは Simulink モデルで実行されるような汎用計算に対応できないため，カスタムカーネル [7] を利用することで汎用計算を OpenVX プログラムでも行えるようにした．このカスタムカーネルでは汎用計算ができ，なおかつハードウェアへの依存性がない OpenCL で記述することにした．

4. コード変換

この章では Simulink モデルから OpenVX および OpenCL 併用コードに変換する手順について述べる．今回使用する Simulink モデルは ForIterator サブシステムをデータ並列箇所に採用していることとする．今回は図 2 で示したようなフローチャートに従って変換を行った．

4.1 OpenVX グラフ作成

OpenVX はグラフ構造を持つプログラムとなっている．そのため，まず Simulink モデルから OpenVX グラフを作成することにした．このとき，データ並列箇所は ForIterator サブシステム単位で区別できるため，OpenVX におけるノードも基本的には ForIterator サブシステム単位で行うことにした．

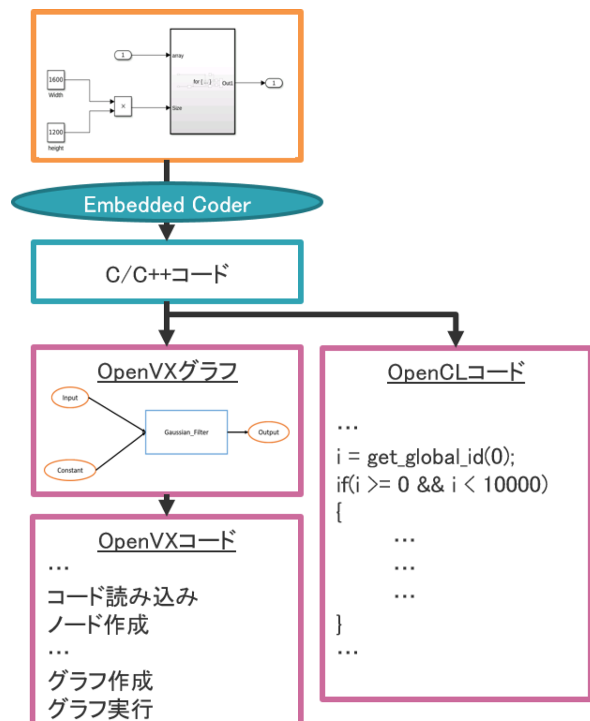

図 2 Simulink モデルから OpenVX 及び OpenCL コードへの変換フローチャート

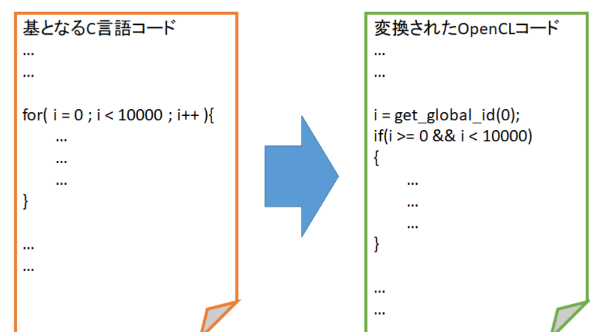

図 3 OpenCL コード変換例

4.2 OpenCL コード作成

カスタムカーネルとして使用する OpenCL コードは図 2 にあるように Simulink モデルから Embedded Coder を用いて作成されたものを使用する．また，OpenCL コードに変換する部分は ForIterator サブシステムとなっている部分なので for ループで記述されている．この for ループにおけるループ単位で並列化を行うことにした．

そこで，図 3 に示したように for(...) となっている部分に着目してコード変換を行った．

内容としては for 命令文で変動する変数をアクセラレータを制御する ID として使用し，OpenCL の get_global_id 関数などを用いて値を設定することにした．このとき，for 命令文を削除した．また，それ以外の部分は基本的にはそのままとした．

© 2020 Information Processing Society of Japan

Vol.2019-ARC-240 No.39
Vol.2019-SLDM-191 No.39
Vol.2019-EMB-53 No.39
2019/2/28

表 1　データ並列が行える Simulink ブロックの比較

	ForIterator サブシステム	Unbuffer・Buffer ブロック	Mux・Demux ブロック
モデル上での並列箇所	サブシステム全体	2 つのブロックの間	2 つのブロックの間
並列単位	1 ループ	取り出す要素数	Demux ブロック依存
並列箇所の ID	ForIterator ブロックに記述	不明	不明
グラフ記述形式	DAG	DAG	DAG
プログラム記述形式	SIMD	SIMD	MIMD
C/C++コード記述形式	for ループ	マルチレート処理	複数個所での処理

表 2　プログラム言語の比較

項目	CUDA	OpenVX	OpenCL
ベース言語	C/C++,Fortran	C/C++	C 言語
対象処理	汎用計算	画像処理	汎用計算
ハードウェアへの依存	NVIDIA 社製の GPU のみ	なし	なし
グラフ構造との相性	×	○	×

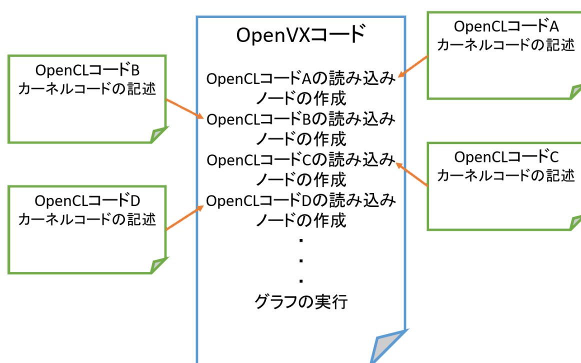

図 4　OpenVX コードと OpenCL コードの関わり

表 3　CPU 実行環境

項目	CPU 実行環境
CPU	Intel(R) Core(TM) i7-6700K
メモリ	8GB DDR4 DIMM × 4 枚

表 4　GPU 実行環境

項目	GPU 併用環境
CPU	Intel(R) Core(TM) i7-6700K
メモリ	8GB DDR4 DIMM × 4 枚
GPU	TITAN X (Pascal) × 2 枚
I/O シリアルインターフェース	PCI Express 3.0 x16
OpenVX	OpenVX1.1
OpenCL	OpenCL1.2 ver2.2.8-1

4.3　OpenVX コード作成

OpenVX コードでは主に OpenCL コードの読み込みとノード作成、グラフの作成と実行を行うことにした．OpenCL コードとのかかわりは図 4 の様になる．

また，OpenVX コードでは変数の初期値設定も行うことにした．

5.　評価

この章では，いくつかの Simulink モデルに対して今回提案した手法を適用し，そのコード変換による実行結果の変化や性能向上に対して評価を行った．

また，同一の Simulink モデルから OpenVX コードを作成する際に複数種類の OpenVX グラフを作成し，それぞれの実行時間の比較も行った．

5.1　評価環境

今回は Simulink モデルから作成された C コードの実行環境として表 3 に示したような CPU 実行環境を，提案手法を適用した OpenVX と OpenCL の併用コードの実行環境として表 4 に示した GPU 併用環境を用意した．

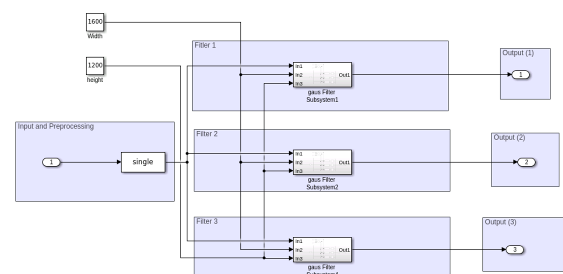

図 5　Filter モデル

5.2　評価 1：モデルの違いによる性能向上の違い

評価 1 では 4 種類の Simulink モデルに対して今回提案した手法を適用し，そのコード変換による実行結果の変化や性能向上に対して評価を行った．

5.2.1　使用モデル

評価 1 で使用するモデルは画像をエッジフィルターに通す Filter モデル（図 5）と Filter モデルのフィルター部分を 3 × 3 ガウスフィルターに変更した Filter_gaussian モデル，MATLAB 上で作成したニューラルネットワークを Simulink モデルに変換した [8][9]NN モデル（図 6），畳み込みニューラルネットワークを Simulink モデル上で再現した CNN モデル（図 7）の 4 種類となる． 評価モデルの

© 2020 Information Processing Society of Japan

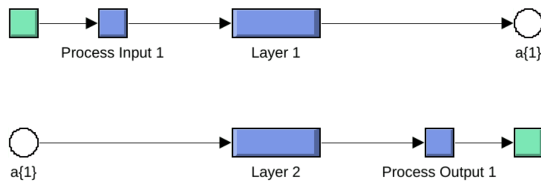

図 6　NN モデル

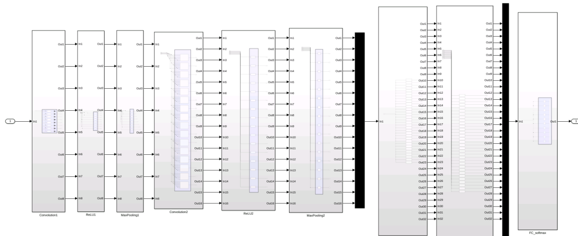

図 7　CNN モデル

実行時に Filter モデルと Filter_gaussian モデルでは画像の大きさを変動させて性能向上の変化を計測した．NN モデルではレイヤーの数を変えずにそれぞれのレイヤーが持つニューラル数を変動させて性能向上の変化を計測した．CNN モデルでは処理を行う画像の大きさを変動させて性能向上の変化を計測した．

5.2.2　評価結果

実行結果は全ての場合において提案手法を適用する前と後で変化は生じなかった．

GPU 併用実行の対 CPU 実行性能向上比をまとめたグラフを図 8 から図 11 に示す．

グラフの横軸はモデルによって異なる値を示している．2 種類のフィルターモデルの結果を表したものでは処理した画像の画素数を，NN モデルでは各レイヤー層のニューラル数を，CNN モデルでは処理した画像の数をそれぞれ表している．

また，グラフの縦軸は全てのモデルで対 CPU 性能向上比を示しており，CPU 実行環境での実行時間 ÷GPU 併用環境での実行時間で求めた値となっている．

グラフからほとんどすべての場合で性能が向上していることが分かる．

5.2.3　考察

提案手法を用いて OpenVX と OpenCL コードへの変換を行うことで性能が向上できた．このとき最大で 117 倍ほどの性能向上となった．同時に CNN モデルにおいては低画素のときには元のコードに比べて遅くなってしまった．このように今回の手法では必ずしも高速化できるわけではないことも分かった．

性能向上が最大で 117 倍ほどとなったのは，今回使用した環境において最大での同時実行スレッド数が 128 個となっていたことが理由として考えられる．

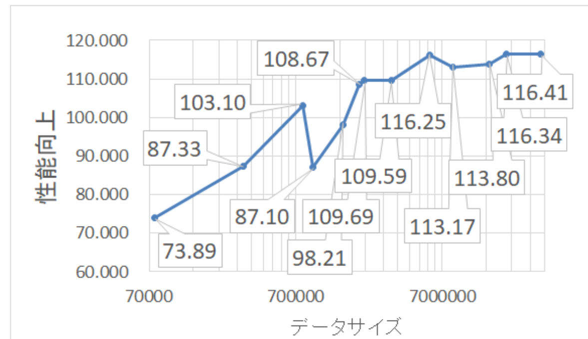

図 8　Filter モデルでの対 CPU 性能向上比

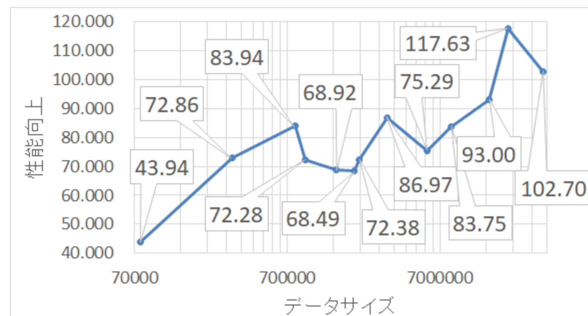

図 9　Filter_gaussian モデルでの対 CPU 性能向上比

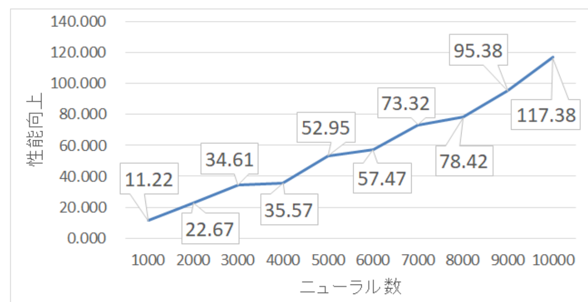

図 10　NN モデルでの対 CPU 性能向上比

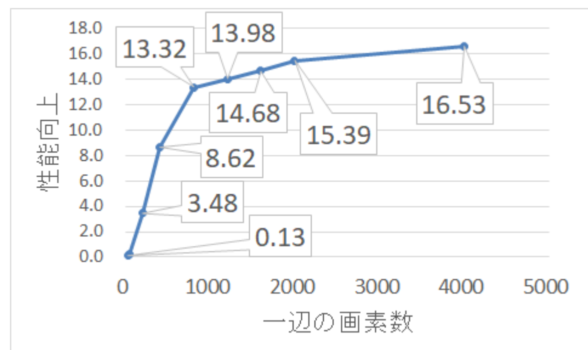

図 11　CNN モデルでの対 CPU 性能向上比

4 種類のモデルのうち CNN モデルだけは性能向上の伸びが極端に悪かったがこれは他のモデルに比べて，使用する画像が小さかったため並列性が低かったことと，シーケンシャルな部分が多いためにその点でも並列性が確保できなかったことが理由として挙げられる．このように今回の

© 2020 Information Processing Society of Japan

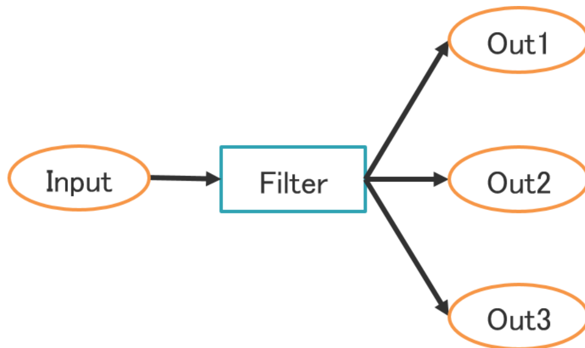

図 12　粒度が粗い OpenVX グラフ

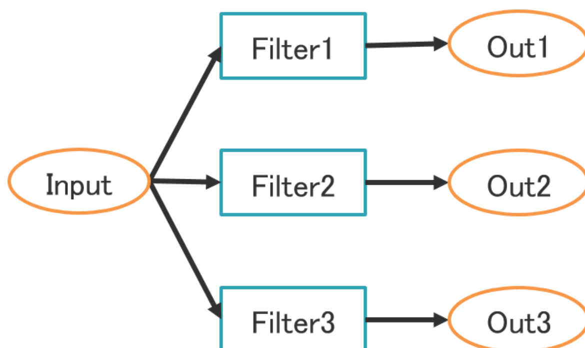

図 13　粒度が中くらいの OpenVX グラフ

手法を適用しても元々のモデルの並列性が低かった場合，性能向上は果たせないことも分かった．

5.3　評価 2

評価 2 では同一の Simulink モデルから複数種類の OpenVX グラフを作成し，それぞれで性能向上に差が生じるかどうかを比較した．

5.3.1　使用モデル

評価 2 で使用したモデルは評価 1 で使用した Filter モデル（図 5）と同じものとなっている．実行に使用した画像は 1600 × 1200 のものとなっている．

5.3.2　作成した OpenVX グラフ

今回作成した OpenVX グラフは図 12 に示した粒度が粗いものと，図 13 に示した粒度が中間のもの，図 14 に示した粒度が細かいものの 3 種類となっている．

3 種類のグラフで総計算量は変動しておらず，実行内容も同一のものとなっている．

5.3.3　評価結果

各 OpenVX グラフを適用した場合における GPU 併用実行の対 CPU 実行性能向上比を図 15 にまとめた．

グラフの違いによって性能向上に大きな差が見られなかった．

5.3.4　考察

グラフによって実行時間に大きな差が生じないという結果になったが，これは評価 1 の考察で述べたように今回使

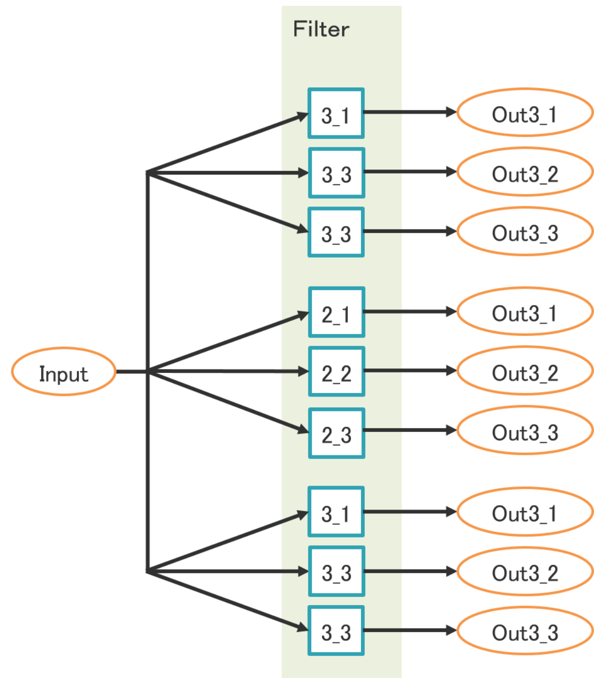

図 14　粒度が細かい OpenVX グラフ

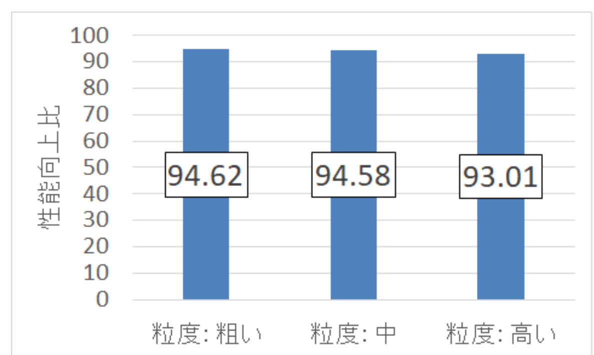

図 15　粒度別の対 CPU 性能向上比

用した環境での最大同時実行スレッド数が 128 個であることが関係してくると考えられる．つまり，今回の環境ではいくら並列性が存在しても最大で 128 倍までしか高速化が果たせず，今回の変更による影響がほとんどなくなってしまった可能性がある．

6.　先行研究

この章では OpenVX を用いた高速化に関する先行研究について述べる．同時にそれらの研究に対する本研究の独自性などについても述べる．

6.1　OpenVX を用いた高速化手法に関する研究

OpenVX を用いた高速化手法としては，メモリアクセス方法を変更することで高速化を図るというものがある [10]．この研究ではメモリアクセスを一定区画ごとに行うタイリングやメモリアクセスを順次行うパイプライン実行といっ

© 2020 Information Processing Society of Japan

た手法よる高速化の可能性について述べている．

他にはグラフ実行のノードを細粒化することで高速化を図るという研究もある [11]．これは OpenVX で実行されるグラフをより細かくし，それぞれをホストとアクセラレータに適切に割り振ることで実行の高速化を果たしたり，応答限界の分析をより厳密に行ったりする研究となっている．

6.2 IoT デバイスに向けた OpenVX 活用に関する研究

IoT デバイス向けの OpenVX 実装手法として，標準の OpenVXAPI と比べて占有メモリが少ない専用の API を作成し，それを利用するという研究もある [12]．これは IoT デバイスで用いられるオンチップのメモリでは使用できるメモリが少なく，標準の OpenVX ライブラリでは実行できなくなるという問題を解決するためのものとなっている．

6.3 先行研究に対する本研究の位置づけ

これらの研究はすでに存在している OpenVX コードの実行を高速化するための手法について研究したものであり，今回研究を行ったような別のプログラム形式で記述されたコードを OpenVX コードに変換することで高速化を図るという研究は少ない．その点で本研究は独自性があると言える．また，本研究はあくまでも OpenVX コードに変換するだけであり，使用する OpenVX ライブラリはほとんどが標準のものとなっている．そのため，先行研究として挙げたものに書かれてある標準 OpenVX ライブラリで記述されたプログラムを高速化する手法と本研究で提案する手法は併用することで更なる高速化が図れると考えられる．

また，評価 2 ではグラフの粒度と実行時間に大きな関係がないといった結果となっており，先行研究とは異なった結論になっている．これは今回分割した部分は全てアクセラレータで実行される部分であり，先行研究にあるようなホストとアクセラレータが両方とも実行される部分を分割したわけではないため，粒度を細かくしても実行時間に影響しないと考えられる．

7. まとめ

本研究ではデータ並列性を利用して高速化を図るモデルベース並列化手法を提案した．これには ForIterator サブシステムを用いた Simulink モデル作成手法と，ForIterator サブシステムが使用されていることを前提とした OpenVX および OpenCL 併用コード生成手法が含まれている．

また，評価 1 で提案手法によって高速化が果たせるかどうかを評価した．その結果最大 117 倍ほどの高速化を果たせることが分かった．この結果はあくまでも今回の環境における結果であるため，他の GPU や DFP，FPGA を用いた環境では高速化が果たせるかどうかは分からない．また，そのような環境ではよりよい性能向上ができる可能性もある．

このとき，CNN モデルでは画素数が小さいときには性能が CPU に比べて低下した．これは画素数が小さいことによるデータ並列性の低下が考えられる．さらには CNN モデル自体がシーケンシャルな部分が多いため，そうした点でもデータ並列性が低下したのだと考えられる．

評価 2 ではグラフによってその性能向上に差が生じるかどうかを評価した．今回の評価ではグラフによる性能向上に大きな差は生じないことが分かった．

8. 今後の課題

今後の課題としてはまず，OpenVX だけでなく，近年環境が広まってきた SYCL[13] などといった他の言語にも対応していきたいと考えている．

さらには今回の提案手法を手動で行ったため，これを自動で適用できるようにするための研究も行っていきたい．

参考文献

[1] MATLAB，MathWorks 社 https://jp.mathworks.com/products/matlab.html?s_tid=hp_ff_p_matlab
[2] Simulink，MathWorks 社 https://jp.mathworks.com/products/simulink.html?s_tid=hp_ff_p_simulink
[3] 次世代半導体 IP「DFP」を搭載したテストチップを開発，NSITEXE 社 https://nsitexe.com/archives/618
[4] CUDA Zone，NVIDIA 社 https://developer.nvidia.com/cuda-zone
[5] OpenVX Overview，Khronos グループ https://www.khronos.org/openvx/
[6] OpenCL Overview，Khronos グループ https://www.khronos.org/opencl/
[7] Custom OpenCL(TM) Kernel in OpenVX* Sample，Intel 社 https://software.intel.com/en-us/sample-custom-opencl-kernel
[8] 浅いニューラル ネットワークの学習，MathWorks 社 https://jp.mathworks.com/help/deeplearning/ref/train.html
[9] 浅いニューラル ネットワークのシミュレーション用の Simulink ブロックの生成，MathWorks 社 https://jp.mathworks.com/help/deeplearning/ref/gensim.html
[10] E. Rainey 他，“Addressing system-level optimization with openvx graphs”, 2014 IEEE Conference on Computer Vision and Pattern Recognition Workshops, 2014, pp. 644-649.
[11] Ming Yang 他，“Marking OpenVX Really“Real Time””, 2018 IEEE Real-Time Systems Symposium (RTSS), 2018, pp. 80-93.
[12] Giuseppe Tagliavini 他，“Enabling OpenVX support in mW-scale parallel accelerators”, 2016 International Conference on Compliers, Architectures, and Sythesis of Embedded Systems (CASES), 2016, pp. 1-10.
[13] SYCL Overview，Khronos グループ https://www.khronos.org/sycl/

© 2020 Information Processing Society of Japan