Vol.2021-ARC-244 No.19
Vol.2021-SLDM-194 No.19
Vol.2021-EMB-56 No.19
2021/3/25

データ並列性を持つSimulinkモデルからのコード生成

徐 品[1]　枝廣 正人[1,a)]

概要：本論文では，S-Function Builder ブロックを用いてデータ並列処理を記述した Simulink モデルから，データ並列化 SYCL コードを生成し，ヘテロジニアス計算機環境で実行する手法を提案する．その手順の大部分をスクリプトで自動化できる．また，そのデータ並列化手法と，我々が提案しているタスク並列化ツールである MBP を併用することができる．評価において，提案手法でデータ並列化したプログラムは逐次プログラムと比べて実行結果に差異がなく，CPU-GPU 計算機上で最大約 547 倍の高速化を達成した．また，タスク・データ並列化を併用したプログラムは，いずれの並列化手法を単独に適用したプログラムより性能が優れることを確認できた．

Code Generation from Simulink Models with Data Parallelism

Abstract: In this paper, we propose a method where data-parallel SYCL code is generated from Simulink models in which computations with data parallelism are expressed in the form of S-Function Builder blocks, and is executed in heterogeneous computing environment. Most parts of the procedure can be automated with scripts. Also, the data-parallel method can be applied together with MBP, another parallelizer proposed by us that exploits task parallelism. In evaluation, data-parallel programs generated using our proposed method achieved a maximum speedup of approximately 547 times compared to sequential programs, without observable difference in the computed results. Also, the programs generated while exploiting both task parallelism and data parallelism were confirmed to have achieved better performance than those generated while exploiting either one of the two.

1. はじめに

Simulink[1] を用いた組込みシステム向けモデルベース開発が普及している．一方，組込みシステムにおいて，マルチコア・メニーコアでの性能に関心が集まりつつある．そのため，Simulink で作成したモデルから並列化されたコードを自動生成することが要求されている．このような動向をふまえ，Simulink でのタスク並列性を持つモデルの作成及びそのようなモデルからの並列化コード自動生成ツールの研究が進んでいる [2][3][4][5].

近年，組込みシステムにデータ並列性を持つ複雑な処理を実行するニーズが高まり，これらの処理に特化される GPU 等のアクセラレータを搭載したヘテロジニアス SoC も市場に出回るようになってきた．そのため，データ並列性を持つ Simulink モデルからヘテロジニアスアーキテクチャを有する計算プラットフォームで動作できるプログラムを生成する研究が求められ，研究されてきている [6] が，Simulink モデルの記述方式や，データ並列処理の実現に使用するプログラミング言語などには課題があり，また，コード生成の自動化及びタスク並列化との併用ができていない．

本研究はデータ並列性を持つ Simulink モデルからのコード生成に対して，[6] とは異なる手法を提案した．具体的には以下の通りである．

(1) Simulink モデルでデータ並列処理を実現する手法として，S-Function Builder ブロック [7] を利用
(2) データ並列処理を実現するために必要な並列プログラミング言語として，SYCL[8] を使用
(3) Simulink モデルを SYCL コードに変換する方法として，Embedded Coder[9]，PPCG[10] 及び Intel DPCT[11] を利用し，CUDA[12] を経由してコードを生成
(4) タスク並列化と併用する手法として，データ並列化の対象とする関数のアクセラレータでの実行プロファイルを取ったうえで，そのアクセラレータを CPU コアと見做してタスク並列化を実施し，その後対象関数を

1 名古屋大学大学院情報学研究科
Furo-cho, Showa-ku, Nagoya, Aichi, 464–8603, Japan
a) eda@ertl.jp

ⓒ 2021 Information Processing Society of Japan

データ並列化.

評価において，提案手法でデータ並列化したプログラムは逐次プログラムと比べて実行結果に差異がなく，CPU-GPU 計算機上で最大約 547 倍の高速化を達成した．また，タスク・データ並列化を併用したプログラムは，いずれの並列化手法を単独に適用したプログラムより性能が優れることを確認できた．

2. 使用ツール

SYCL[8] は Khronos グループが標準化した抽象化中間レイヤーであり，標準の C++文法に従うコードを用いてマルチプラットフォーム対応のヘテロジニアス処理を目標としている．

SYCL を様々なターゲットプラットフォームにて実装するプロジェクトが複数存在する．そのなかで，Intel DPC++ は SYCL 標準に基づいて C++を拡張したプログラミング言語で，SYCL 標準が規定した機能及び Intel 社独自の機能を有する．

Intel OneAPI Base Toolkit[13] は Intel 社が Intel DPC++を中心として提供しているツールキットであり，前述の機能を CPU，GPU 及び Intel 製 FPGA といった多様なターゲットプラットフォームにて実現することができる．そのツールキットを構成する様々なツールのなかで，本研究において主に使うものは

- Intel oneAPI DPC++/C++ Compiler, Clang/LLVM に基づいた Intel DPC++のオープンソースのコンパイラ
- Intel DPC++ Compatibility Tool (Intel DPCT), CUDA で書かれた既存のコードを Intel DPC++に変換するツール

といった二つである．

PPCG (Polyhedral Parallel Code Generator) [10] は並列化プログラミングにおける最適化モデルである Polyhedral モデル [14] に基づいて，Verdoolaege らが開発したソース・ツー・ソースコンパイラである．PPCG は，C 言語で書かれた，一定の条件を満たす多重 for ループからデータ並列操作を抽出し，それと同等の処理を行う CUDA コードを生成する機能を有する．

3. 先行研究

3.1 BLXML

BLXML (Block Level XML) は我々が提案した XML 記述であり [15]，ブロック線図の構造とそのブロックに基づいて Embedded Coder が生成した C/C++コードのほか，ブロック間の依存関係，ブロックの実行にかかる時間や，タスク並列化コードにおけるブロックのコア配置などといった情報を保持することができる．

3.2 モデルベース並列化ツール（MBP）

我々は Simulink モデルからタスク並列化コードを自動生成する手法に関する研究 [2][3] を行い，モデルベース並列化ツール（Model Based Parallelizer，以下，MBP）を開発した．その成果の一部は [4] に反映されている．

MBP におけるタスク並列化コード生成の手順を図 1 に示す．以下，その手順の各ステップを紹介する．

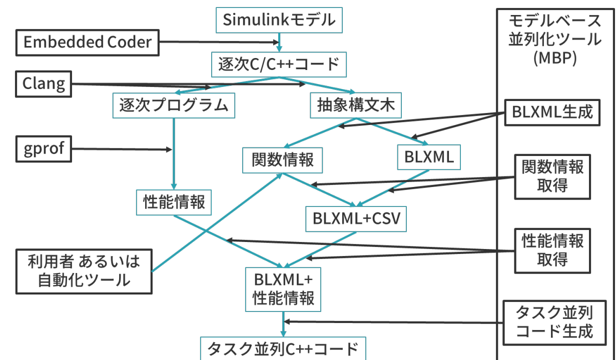

図 1 MBP におけるコード生成の手順

3.2.1 BLXML 生成

MBP においては，まず Embedded Coder[9] で逐次コードを生成し，それに Clang を使って JSON 形式の抽象構造木を抽出する．その後，この逐次コードと JSON ファイルから，MBP が BLXML を生成する．

3.2.2 関数情報取得

Embedded Coder が常に生成する関数，及び利用者が指定する他の関数に対して，逐次実行が求められることがある．MBP は各関数名とその関数がソースコードに占める行の範囲を CSV ファイルにまとめ，逐次実行の指示をそのファイルに反映する．

3.2.3 性能情報取得

MBP はターゲットシステムが提供するプロファイラ（GNU/Linux 環境においては gprof[16]）を使って逐次プログラムの性能情報をフラットプロファイル形式で保存し，そのプロファイルを読み込んで BLXML の対応箇所に追加する．

3.2.4 タスク並列化コード生成

タスク並列化コードの生成は，

(1) 各ブロックを性能情報に基づいて，[2] の手法を用いてターゲットシステムの CPU コアに割当てる
(2) コア配置の結果に基づいて，逐次コードを C++の thread ライブラリを用いた並列化コードに変換する

といった二つのステップで行われる．

3.3 データ並列性を持つモデルからのコード生成

甲斐らは [6] にて，

(1) Simulink が提供する For-Iterator サブシステムを利用し，データ並列性がある操作を Simulink モデルに記

述する

(2) 並列プログラミング言語として，OpenVX[17] と OpenCL[18] を併用する

(3) 手作業を前提に, Simulink モデルを OpenVX・OpenCL 併用コードに変換する

といった手順でデータ並列化コードを生成することを提案した．その手順を図 2 にまとめる．

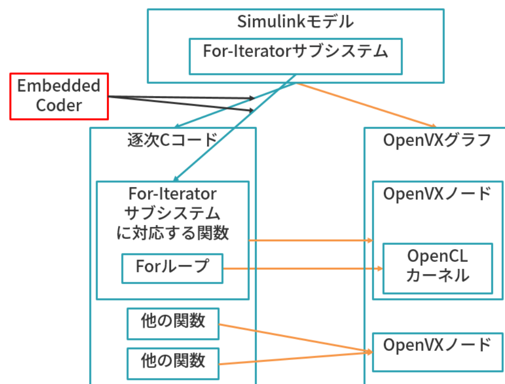

図 2 [6] が提案したデータ並列化コード生成の手順

3.4 従来手法に対する本研究の位置づけ

[6] のコード生成手法には，記述できる処理の範囲が限定的であること，Simulink モデルが複雑になりやすいこと，及び複数の並列プログラミング言語の使用により自動化もしくは MBP との併用が複雑であることといった課題がある．それに対して，本研究は [6] と異なるデータ並列化コード生成手法を提案し，それらの課題の解決を目指す．

4. 提案手法

4.1 データ並列化コード生成手法

4.1.1 Simulink モデル記述法

簡潔な Simulink モデルに多様な処理を記述するために，利用者が自ら設計することができるカスタムブロックを利用してデータ並列処理を記述することが効率的である．Simulink がサポートするカスタムブロックの特性比較を表 1 にまとめる．

S-Function Builder ブロックには状態を保持できるため，多様な処理を表せる．その上，S-Function Builder ブロックに入力されたコードはそのまま Embedded Coder が生成した逐次コードに転記される．そのため，Simulink モデルを作成する段階で，適切な箇所にコメントもしくはコンパイラディレクティブを追加することで，データ並列化箇所を示すコンパイラディレクティブを逐次コード上に残すことが可能である．さらに，TLC ファイルを自動生成できるため，コード生成手順から手動作業を減らすことができる．

以上の比較を踏まえ，S-Function Builder ブロックを採用した．

4.1.2 並列プログラミング言語

データ並列処理を実現するために必要な並列プログラミング言語として，SYCL[8] を使用することを提案する．また，SYCL の実装に関しては，Intel DPC++[13] を採用する．

表 2 は本研究が採用しうる並列プログラミング言語の特性の違いを示す．

SYCL には汎用的な演算に利用されることを想定していること，ホストとターゲットで実行されるコードが同じ文法で書かれること，及び抽象度の高さによって同じコードが複数の計算機環境にて実行できることといった利点があり，さらに対応するターゲットプラットフォームが多様であることもデータ並列化コード生成に相応しいため，SYCL を採用することにした．

オープン標準である SYCL を実装するプロジェクトは数多く存在する．表 3 は 2021 年 1 月現在の，主なプロジェクトの状態とサポートするプラットフォームの比較をまとめたものである [13][19][20][21]．

オープンソースである Intel DPC++は正式にリリースされ，積極的に保守されており，サポートするターゲットプラットフォームが多いことを鑑み，SYCL の実装として採用した．

4.1.3 コード変換手法

Simulink モデルを SYCL コードに変換する方法として，以下の手順を提案する．

(1) Embedded Coder[9] を使用し，前述の手法でデータ並列処理を入力した S-Function Builder ブロックを含む Simulink モデルを C コードに変換する．

(2) PPCG[10] を使用し，その C コードを CUDA コードに変換する．

(3) Intel DPCT[13] を使用し，その CUDA コードを SYCL コードに変換する．

C コードに基づいて他言語コードを生成をするために，一般的には Clang/LLVM などのツールでその C コードを LLVM-IR などの低水準中間表現に変換し，解析することが必要である．しかし，SYCL は C++と同水準言語であるため，低水準中間表現から生成することは難しい．

一方，Intel DPC++のコンパイラを含む Intel oneAPI Base Toolkit は，CUDA コードを機械的に SYCL コードに変換する機能を有する Intel DPCT を提供している．そのため，CUDA を中間表現として使用することで，低水準言語を扱うことなく，SYCL コードへの簡便な変換を図ることができる．

CUDA を中間表現として使用するには，C コードを CUDA コードに変換できるソース・ツー・ソースコンパイラが必要である．しかし，そのようなコンパイラは必ずしも C 言語で書かれたコードをすべて変換できるわけではな

ⓒ 2021 Information Processing Society of Japan

表 1　カスタムブロックの比較

	S-Function Builder	Level-2 MATLAB S-Function	C-MEX S-Function	MATLAB Function	C Caller
性能	高速	高速	高速	低速	高速
内部状態	あり	あり	あり	なし	なし
入力言語	C/C++	MATLAB	C-MEX	MATLAB	C
TLC 作成	自動	手動	手動	N/A	N/A

表 2　並列プログラミング言語の比較

	SYCL	OpenVX	OpenCL	CUDA
利用想定	汎用的	画像処理	汎用的	汎用的
使用言語	C++	C/C++	C/C++	C/C++
ホストとカーネルの文法	同じ	違う	違う	違う
抽象度	高い	高い	低い	低い
ターゲットプラットフォーム	多様	多様	多様	Nvidia GPU
オープン標準	◯	◯	◯	×

い．その上，本研究はコード生成の自動化を目指すため，コード生成の途中に発生しうるエラーをできるだけ削減することが重要である．そのため，本研究が利用するソース・ツー・ソースコンパイラには，

(1) 変換できる処理の範囲ができるだけ広いこと
(2) 変換する対象を前述のように，コメントもしくはコンパイラディレクティブで指定された並列化箇所に限定できること

といった二つの条件が求められる．PPCG はこれらの条件を満たすことを鑑み，その採用を提案する結論に至った．

4.2　データ並列化とタスク並列化の併用

タスク並列化と併用する手法として，データ並列化の対象とする関数のアクセラレータでの実行プロファイルを取ったうえで，そのアクセラレータを CPU コアと見做し，MBP を用いてタスク並列化を行い，その後，その関数の記述をデータ並列化された SYCL コードに置き換えることを提案する．

MBP はヘテロジニアスマルチコアのアーキテクチャを有するターゲットプラットフォームに対応している．そのため，SYCL コードを実行するアクセラレータを CPU コアの一種類と見做すことは MBP の使用に支障をきたさない．その上，MBP は std::thread ライブラリを用いたマルチコア C++プログラムを生成する機能を有する．従って，別スレッド実行するようになったとしても，関数の本体を記述するコードには変更の必要がない．そのため，データ並列化対象とする関数を，変換された SYCL コードに置き換えることができる．

4.3　コード生成の自動化

4.1.3 節で述べたデータ並列化コード生成手順を図 3 に示す．そのなかで，逐次コードの生成は Embedded Coder を用いて手動で行わなければならないが，CUDA コード生成，SYCL コード生成，及びデータ並列化プログラムのコンパイルについては，本研究により自動化を実現した．

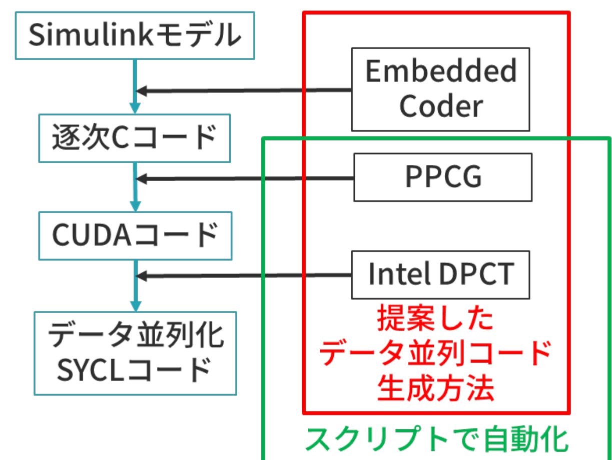

図 3　提案したデータ並列化コード生成手順

4.2 節で述べた手法を用いて，タスク・データ並列化併用コードを生成する手順を図 4 に示す．そのなかで，データ並列化プログラムの生成は前述のとおり，手動で行わなければならないステップを含むが，それ以外の各ステップの自動化は本研究により実現した．

5.　評価

本章では，提案したデータ並列化コード生成手法とタスク・データ並列化併用コード生成手法により生成したプログラムの正確性，及び逐次コードに対する速度向上に対する評価実験について述べる．

5.1　実験方法

並列化プログラムと逐次プログラムの実行結果における差異を確認し，提案手法を用いて生成したデータ並列化

表 3　SYCL を実装するプロジェクトの比較

	Intel DPC++	triSYCL	hipSYCL	ComputeCpp
主導者	Intel	Xilinx	ハイデルベルク大学	Codeplay
リリース種類	正式	未完成	ベータ	正式
最近のリリース	2020 年 12 月	2018 年 3 月	2020 年 12 月	2020 年 11 月
オープンソース	◯	◯	◯	×
CPU	◯	◯	◯	◯
GPU	Intel/NVIDIA	NVIDIA	NVIDIA	Intel/AMD/NVIDIA
FPGA	Intel	Xilinx	×	×

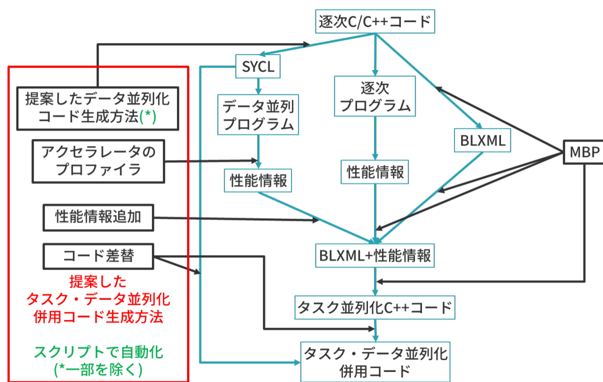

図 4　提案したタスク・データ並列化併用コード生成手順

コードの正確性を評価する．また，各プログラムにおいて，Simulink モデルが表す処理を実行する関数を 100 回繰り返して実行し，平均して 1 回の実行に要する時間を当該プログラムの性能を示す平均実行時間とし，逐次プログラムと並列化プログラムの平均実行時間の比の値を当該並列化手法による高速化倍数と定義する．

5.2　評価環境

この章の実験において，逐次プログラムを実行する環境を表 4 に示す．データ並列化プログラムを実行する環境に関しては，マルチコア CPU 環境を表 5 に，CPU+GPU 環境を表 6 に示す．タスク・データ並列化併用プログラムを実行する環境は表 6 に示した CPU+GPU 環境である．

表 4　逐次プログラムを実行する環境

項目	内容
CPU	Intel Core i9-7900X (3.30GHz) (10 コア 20 スレッド)
メモリ	16GB DDR4 DIMM×8
C 言語	C11
C コンパイラ	Clang 12.0.0 (trunk)

表 5　マルチコア CPU 環境

項目	内容
CPU	Intel Core i9-7900X (3.30GHz)
メモリ	16GB DDR4 DIMM×8
SYCL コンパイラ	Intel oneAPI DPC++ Compiler 2021.1
SYCL バックエンド	Intel OpenCL 2.1

表 6　CPU+GPU 環境

項目	内容
CPU	Intel Core i9-7900X (3.30GHz)
メモリ	16GB DDR4 DIMM×8
GPU	Nvidia Titan V (1.20GHz)
SYCL コンパイラ	Intel oneAPI DPC++ Compiler 2021.1
SYCL バックエンド	Nvidia CUDA 10.2.89

5.3　評価 1：データ並列化による速度向上

評価 1 では，提案手法を用いて生成したデータ並列化コードから，マルチコア CPU 環境及び CPU+GPU 環境向けにコンパイルしたデータ並列化プログラムを，Embedded Coder で生成した逐次 C コードからコンパイルした逐次プログラムと比較し，実行結果の差異及び高速化の効果を評価する．

5.3.1　使用するモデル

評価 1 では，簡単なベクトル計算を行うモデル（図 5），画像のガウスフィルタ処理と差分を計算するモデル（図 6），及び行列の掛け算を行うモデル（図 7）といった 3 種類のモデルを使用する．なお，ベクトル計算におけるベクトル長は 10000〜50000000，ガウスフィルタ処理における画像の画素数は 76800〜64000000，行列計算における行列サイズは 250 × 250〜5000 × 5000 とした．

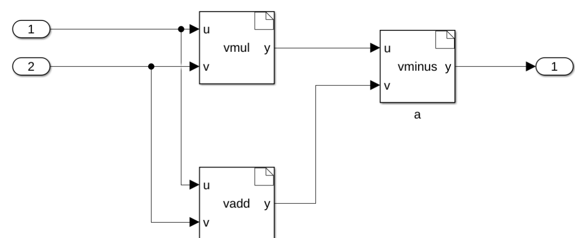

図 5　簡単なベクトル計算モデル

ⓒ 2021 Information Processing Society of Japan

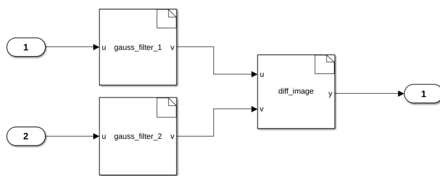

図 6　ガウスフィルタ差分モデル

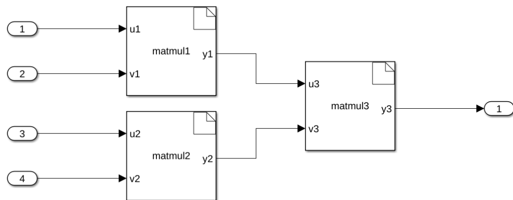

図 7　行列の掛け算モデル

5.3.2　評価結果

すべてのモデルにおいて，生成したデータ並列化プログラムと逐次プログラムは，実行結果に差異がなかった．これによって，評価の範囲においては，提案手法を用いて生成したデータ並列化プログラムは正確に演算を実行していることがわかった．

各モデルにおいて，マルチコア CPU 環境及び CPU+GPU 環境にて，データ並列化による高速化倍数を計測した結果を図 8 から図 11 に示す．

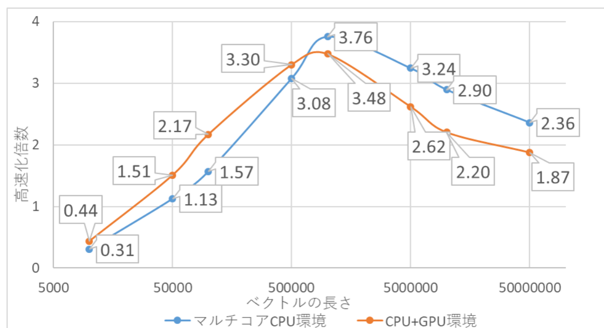

図 8　簡単なベクトル計算モデルにおける高速化倍数

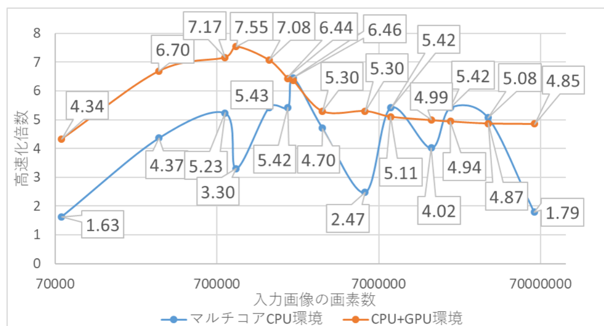

図 9　ガウスフィルタ差分モデルにおける高速化倍数

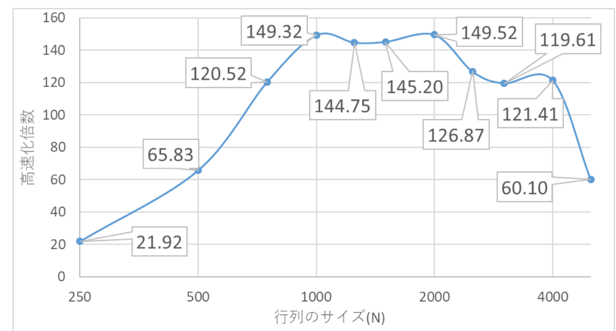

図 10　行列の掛け算モデルにおける高速化倍数（マルチコア CPU 環境）

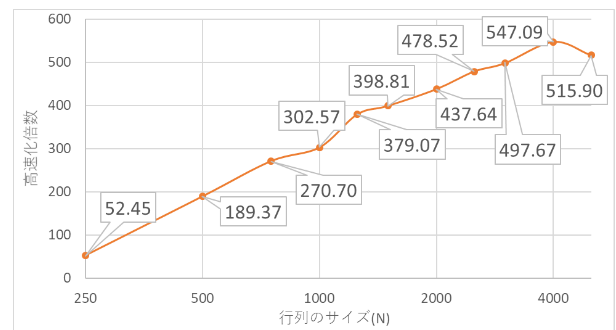

図 11　行列の掛け算モデルにおける高速化倍数（CPU+GPU 環境）

5.3.3　考察

5.3.3.1　モデルの違いによる高速化の効果の差異

評価 1 に使用した 3 種類のモデルと 2 種類の実行環境において，すべて実行結果が正しく，データ並列化による高速化を達成したが，異なるモデルにおいて高速化の効果には大きな差異があった．モデル，環境別に達成した最大高速化倍数を表 7 にまとめた．

表 7　モデル，環境別に達成した最大高速化倍数

	マルチコア CPU	CPU+GPU
簡単なベクトル計算モデル	3.76	3.48
ガウスフィルタ差分モデル	6.46	7.55
行列の掛け算モデル	149.52	547.09

この差異が存在する原因としては，データ並列化対象とする処理において，並列処理が全体の処理に占める割合が違うことが想定される．

5.3.3.2　マルチコア CPU 環境における高速化の効果

マルチコア CPU 環境に使用したプロセッサのコア数は 10 であり，スレッド数は 20 であるが，行列の掛け算モデルおいては，データ並列化による最大高速化倍数は 149.52 倍であり，コア数とスレッド数を大幅に超える状況であった．この状況になった原因は，PPCG が生成した CUDA コード，及び Intel DPCT がそれに基づいて生成した SYCL コードにはブロック化とアンローリングといった高速化手法が施されたためと推測される．

5.4 評価 2：タスク・データ並列化併用による速度向上

評価 2 では，タスク・データ並列化併用プログラムを，逐次プログラム，MBP が生成したタスク並列化プログラム，及び 4.3 節で述べた手法で生成したデータ並列化プログラムと比較し，実行結果の差異と高速化の効果を評価する．

5.4.1 使用するモデル

評価 2 に使用するモデル（図 12）は行列の掛け算と足し算を行う．データ並列化対象とするブロックを 1 つのみとし，そのブロックの演算規模及び入力，出力のサイズを調整できるようにしている．図の中の matmul ブロックがデータ並列化対象であり，行列サイズは 500 × 500〜2500 × 2500 としている．

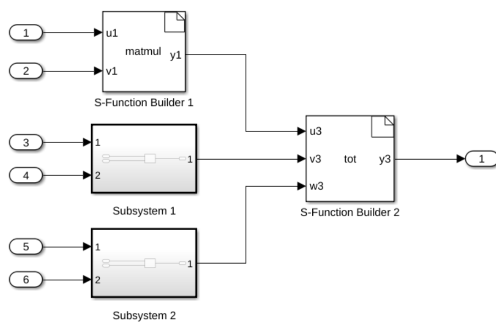

図 12 評価 2 に使用する Simulink モデル

5.4.2 評価結果

生成したタスク並列化プログラム，データ並列化プログラム及びタスク・データ並列化併用プログラムは，逐次プログラムと比較して実行結果に差異がなかった．これにより，評価の範囲においては，提案した MBP との併用手法を用いて生成したタスク・データ並列化併用プログラムは正確に演算を実行していることがわかった．

タスク並列化，データ並列化，及びタスク・データ並列化併用による高速化倍数を計測した結果を図 13 に示す．

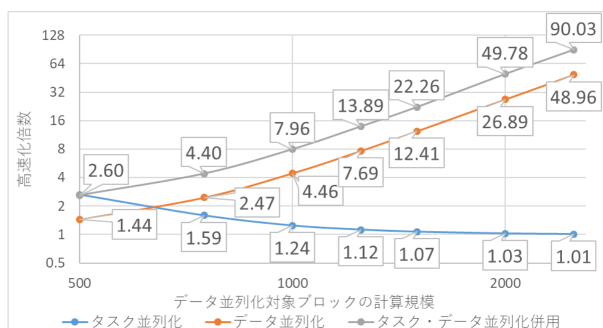

図 13 異なる並列化手法による高速化倍数

5.4.3 考察

タスク・データ並列化併用による高速化倍数は常にデータ並列化を単独に適用した場合よりも高いことがわかった．この結果により，当該モデルに対しては，タスク・データ並列化を併用する手法で生成したプログラムの性能は，タスク，データいずれかの並列化手法を単独に適用して生成したプログラムより優れることが確認できた．

なお，タスク並列化による高速化倍数は，データ並列化対象ブロックにおける問題規模の拡大によって減少するが，データ並列化，並びにタスク・データ並列化併用による高速化倍数はその問題規模の拡大とともに増加することがわかった．この違いの原因は，データ並列化により，データ並列化対象ブロックに対応する処理の所要時間が大幅に削減され，その処理がプログラム全体の実行におけるクリティカルパスを構成しなくなったためであると推測する．この原因は，データ並列化プログラム，及びタスク・データ並列化併用プログラムの平均実行時間において，問題規模の増大による変化が極めて小さいことから確認できる．

6. まとめと今後の課題

本研究はデータ並列性を持つ Simulink モデルからのコード生成に対して，S-Function Builder ブロックを用いてデータ並列性を持つ Simulink モデルを作成し，そのモデルから SYCL を用いたデータ並列化コードを半自動的に変換する方法を提案した．また，その方法と従来のタスク並列化ツールである MBP とを併用する手法を提案した．

評価 1 では提案手法を用いて生成したデータ並列化プログラムの正確性及びデータ並列化による高速化の効果をマルチコア CPU 環境及び CPU+GPU 環境といった 2 種類の環境で検証した．その結果，データ並列化プログラムの実行結果に差異がなく，最大約 547 倍の高速化を達成した．

さらに，評価 2 では提案したタスク・データ並列化を併用する手法を検証し，その手法を用いて生成したプログラムは実行結果が正しく，タスク，データいずれかの並列化手法を単独に適用したプログラムより性能が優れることが確認できた．

今後の課題として，複数のマルチコア CPU が存在する環境，もしくは CPU と複数の GPU が存在する環境など，異なる構成に対し，提案手法の有効性を評価する必要がある．

それに，提案したタスク・データ並列化併用手法には，データ通信に要する時間を計測しておらず，コア割当の根拠である性能情報には反映していない．ヘテロジニアス環境において，この通信時間を正確に見極める手法の提案も課題である．

参考文献

[1] Mathworks: Simulink, The MathWorks, Inc. (online), available from ⟨https://www.mathworks.com/products/simulink.html⟩ (accessed 2021-01-24).
[2] Zhong, Z. and Edahiro, M.: Model-Based Parallelizer for Embedded Control Systems on Single-ISA Heterogeneous Multicore Processors, *2018 International SoC Design Conference (ISOCC)*, IEEE, pp. 117–118 (2018).
[3] 山口滉平，竹松慎弥，池田良裕，李　瑞徳，鍾　兆前，近藤真己，枝廣正人：Simulink モデルからのブロックレベル並列化，組込みシステムシンポジウム 2015 論文集，Vol. 2015, pp. 123–124 (2015).
[4] eSOL: Model Based Parallelizer, eSOL Co.,Ltd. (online), available from ⟨https://www.esol.com/embedded/mbp.html⟩ (accessed 2021-01-24).
[5] Hottger, R., Krawczyk, L. and Igel, B.: Model-Based Automotive Partitioning and Mapping for Embedded Multicore Systems, *International Journal of Computer and Systems Engineering*, Vol. 9, No. 1, pp. 268–274 (2015).
[6] 甲斐琢朗，森　裕司，枝廣正人：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，研究報告組込みシステム (EMB)，Vol. 2020, No. 39, pp. 1–6 (2020).
[7] Mathworks: S-Function Builder, The MathWorks, Inc. (online), available from ⟨https://www.mathworks.com/help/simulink/slref/sfunctionbuilder.html⟩ (accessed 2021-01-24).
[8] Khronos: SYCL, The Khronos Group Inc. (online), available from ⟨https://www.khronos.org/sycl/⟩ (accessed 2021-01-24).
[9] Mathworks: Embedded Coder, The MathWorks, Inc. (online), available from ⟨https://www.mathworks.com/products/embedded-coder.html⟩ (accessed 2021-01-24).
[10] Verdoolaege, S., Carlos Juega, J., Cohen, A., Ignacio Gomez, J., Tenllado, C. and Catthoor, F.: Polyhedral parallel code generation for CUDA, *ACM Transactions on Architecture and Code Optimization (TACO)*, Vol. 9, No. 4, pp. 1–23 (2013).
[11] Intel: Intel DPC++ Compatibility Tool Developer Guide and Reference, Intel (online), available from ⟨https://software.intel.com/content/www/us/en/develop/documentation/intel-dpcpp-compatibility-tool-user-guide/top.html⟩ (accessed 2021-01-24).
[12] NVIDIA: CUDA Toolkit, NVIDIA Corporation (online), available from ⟨https://developer.nvidia.com/cuda-toolkit⟩ (accessed 2021-01-24).
[13] Intel: Intel OneAPI, Intel (online), available from ⟨https://software.intel.com/content/www/us/en/develop/tools/oneapi.html⟩ (accessed 2021-01-24).
[14] Feautrier, P.: Automatic parallelization in the polytope model, *The Data Parallel Programming Model*, Springer, pp. 79–103 (1996).
[15] Honda, K., Kojima, S., Fujimoto, H., Edahiro, M. and Azumi, T.: Mapping Method of MATLAB/Simulink Model for Embedded Many-Core Platform, *2020 28th Euromicro International Conference on Parallel, Distributed and Network-Based Processing (PDP)*, IEEE, pp. 182–186 (2020).
[16] GNU: GNU gprof, Free Software Foundation, Inc. (online), available from ⟨http://sourceware.org/binutils/docs/gprof/⟩ (accessed 2021-01-24).
[17] Khronos: OpenVX, The Khronos Group Inc. (online), available from ⟨https://www.khronos.org/openvx/⟩ (accessed 2021-01-24).
[18] Khronos: OpenCL, The Khronos Group Inc. (online), available from ⟨https://www.khronos.org/opencl/⟩ (accessed 2021-01-24).
[19] Codeplay: ComputeCpp, Codeplay Software Ltd. (online), available from ⟨https://developer.codeplay.com/products/computecpp/ce/home/⟩ (accessed 2021-01-24).
[20] triSYCL: triSYCL, Github, Inc. (online), available from ⟨https://github.com/triSYCL/triSYCL⟩ (accessed 2021-01-24).
[21] Alpay, A.: hipSYCL, Heidelberg University (online), available from ⟨https://github.com/illuhad/hipSYCL⟩ (accessed 2021-01-24).

ⓒ 2021 Information Processing Society of Japan