C-002

異種プロセッサ環境における並列プログラミング手法の提案
Proposal of Parallel Programming Method for Heterogeneous Processing Environment

林 卓哉 † 渡邊 誠也 † 名古屋 彰 †
Takuya Hayashi Nobuya Watanabe Akira Nagoya

1. はじめに

近年，マルチコア CPU や GPU (Graphics Processing Unit) などの異種のプロセッサが混在する計算機環境が一般的になってきており，それらのプロセッサを活用したヘテロジニアスコンピューティングが注目されている．このような環境向けのフレームワークとしては OpenCL (Open Computing Language) [1] があり，動作プロセッサの種別に依存せずにプログラミング可能という特徴をもつ．この特徴を活かし，データ並列可能な処理を行う際に，その処理データを複数の異種プロセッサに分配して並列処理を行うこと（以降，本稿ではこれを「マルチデバイス処理」と呼ぶ）が可能である [2] [3]．このとき，使用可能なプロセッサを無駄なく活用して処理を行うことが求められ，我々は文献 [4] で複数チップを用いたマルチデバイス処理における効率的な負荷分散手法について報告した．しかし，OpenCL では本来行いたい処理以外で多くの記述が必要なこともあり，その負荷分散手法を用いて効率的なマルチデバイス処理を行う OpenCL コードを完成させるまでの設計効率は高くない．

そこで本稿では，文献 [4] の負荷分散手法を内包する並列プログラミング手法として，並列処理を明示的に記述可能な言語記述から，与えられた計算機環境において最適なマルチデバイス処理を実現する OpenCL コードを自動生成する手法を提案する．

2. 背景

2.1 GPU

GPU とは画像処理を高速に行うために設計された演算装置である．近年はその並列演算性能の高さから，GPU を画像処理以外の演算にも活用する GPGPU (General-Purpose computation on GPUs) が注目されている．

GPU は，CPU と同一チップ上に搭載される IGP (Integrated Graphics Processor) と，チップが独立している dGPU (descrete GPU) に分類される．dGPU は数千のプロセッサコアを有しており，それらに低遅延でデータを供給するために，広帯域のメモリが接続される．また，CPU−dGPU 間のデータ転送には一般に I/O バスが用いられ，それを介したデータ転送を要することがオーバーヘッドとなりやすい．一方 IGP は，有するプロセッサコアが数百程度と dGPU と比べて少ない．しかし，CPU と主記憶を共有しているため，プロセッサ間のデータ転送を必要とせずに協調処理が可能である．

2.2 OpenCL

OpenCL は，非営利団体 Khronos Group を中心に仕様の標準化作業が行われており，AMD や Intel，NVIDIA といった主要なプロセッサベンダも標準化グループのメンバーに含まれている．マルチコア CPU や GPU，DSP (Digital Signal Processor) といったプロセッサが OpenCL で使用可能である．

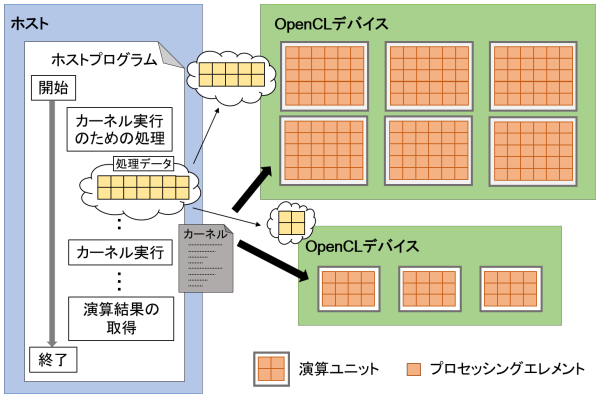

図 1: OpenCL を用いたマルチデバイス処理の流れ

OpenCL では，制御用プロセッサであるホストと演算用プロセッサである OpenCL デバイスの 2 種類の計算要素から構成されるプラットフォームを想定しており，ホストで動作するプログラムをホストプログラムと呼び，OpenCL デバイスで動作するプログラムをカーネルと呼ぶ．また，OpenCL デバイスは複数の演算ユニットから構成され，各演算ユニットはさらに複数のプロセッシングエレメントから構成される．カーネル実行時にそれらの演算要素にインデックスを付与し，各インデックスに対応する処理をカーネルで記述することで，データの各要素を並列に処理する手段を提供している．

図 1 に，OpenCL を用いたマルチデバイス処理の流れを示す．ホストプログラム中の，カーネル実行のための処理には以下が含まれる．

1) プラットフォームの特定
2) OpenCL デバイスの特定
3) コンテキストの作成
4) コマンドキューの作成
5) メモリオブジェクトの作成とデータ書き込み
6) プログラムオブジェクトの作成
7) カーネルのコンパイル
8) カーネルオブジェクトの作成
9) カーネル引数の設定

マルチデバイス処理ではまず，5) の「メモリオブジェクトの作成とデータ書き込み」で，使用する OpenCL デバイスのそれぞれに処理データを分配する．そして各 OpenCL デバイスで，分配された処理データに対してカーネルで記述された処理を実行する．このとき，用意するカーネルは使用する OpenCL デバイスの数に関係なく 1 つだけで良い．最後に各 OpenCL デバイスから演算結果を取得し，処理全体が終了する．

なお OpenCL では，異なるプロセッサ間でプログラムの再利用を可能にするために，使用する OpenCL デバイス毎に上記の 1)〜9) の手続きを明示的に記述する必要がある．各手続きの詳細についてはここでは述べないが，これらは OpenCL が提供するランタイム API を用いて記述可能である．

†岡山大学大学院自然科学研究科, Graduate School of Natural Science and Technology, Okayama University

3. マルチデバイス処理における負荷分散手法

マルチデバイス処理において，より高い性能を得るには，各 OpenCL デバイスへの負荷分散を適切に行う必要がある．そこで本稿ではまず，マルチデバイス処理における負荷分散手法として，カーネル実行時間に基づくシンプルな手法とそれを改良した手法でマルチデバイス処理をそれぞれ行い，その効果について述べる．

3.1 カーネル実行時間に基づく手法

まず，マルチデバイス処理の効果が十分に得られると思われる手法として，アプリケーションを 1 つの OpenCL デバイスで実行（以降，本稿ではこれを「シングルデバイス処理」と呼ぶ）した際のカーネル実行時間をもとにした負荷分散を試みる．マルチデバイス処理で使用する OpenCL デバイスの数を N，k 番目の OpenCL デバイスのカーネル実行時間を T_k とすると，マルチデバイス処理時に k 番目の OpenCL デバイスへ分散する負荷の比 r_k は次の式をもとに算出する．

$$r_k = \frac{1}{\sum_{i=1}^{N} T_k/T_i} \qquad (k = 1, 2, ..., N) \qquad (1)$$

この比 r_k を用いて処理データを各 OpenCL デバイスへ分配し，マルチデバイス処理を行う．

3.2 データ転送時間等を考慮した改良手法

3.1 の手法のみでは，次の 2 点の理由から，無駄の少ないマルチデバイス処理とはならない場合があると想定される．

- 分散負荷の計算時に，ホスト–OpenCL デバイス間のデータ転送時間を考慮していない
- CPU と IGP を用いたマルチデバイス処理時に，CPU コアの 1 つが IGP の制御に割り当てられる

そこで改良手法として，カーネル実行時間に加えてホスト–OpenCL デバイス間のデータ転送時間を考慮した負荷分散を試みる．つまり，式 (1) の T_k をカーネル実行時間とデータ転送時間の和として負荷分散比 r_k を算出する．また，CPU と IGP を使用したマルチデバイス処理時には，CPU コアの 1 つがカーネルを実行できなくなることを考慮し，CPU のカーネル実行時間を (コア数)/(コア数 −1) 倍として負荷分散比を算出する．

以降では，カーネル実行時間に基づく手法を改良前の手法とし，ホスト–OpenCL デバイス間のデータ転送時間等を考慮した手法を改良後の手法とする．

3.3 負荷分散手法の評価

改良前，改良後の各手法を評価するため，特徴の異なるアプリケーションに対してそれぞれの手法でマルチデバイス処理を行い，その効果を比較する．

3.3.1 実行アプリケーションとその実行環境

表 1 に実行アプリケーションとその詳細を示し，表 2 にアプリケーションの実行環境を示す．アプリケーションの実行環境について，マルチコア CPU はホストとして使用すると同時に OpenCL デバイスとしても使用する．そのため，マルチデバイス処理時の OpenCL デバイスの組合せは，(1) CPU+dGPU，(2) CPU+IGP，(3) dGPU+IGP，および (4) CPU+dGPU+IGP の 4 パターンである．

表 1: 実行アプリケーションとその詳細

アプリケーション	特徴	備考
行列積	スレッド間の同期が必要	・4096 × 4096 の行列積 ・行列の各要素は float 型
Canopy クラスタリング	ホスト–OpenCL デバイス間のデータ転送が複数回発生	・10^7 個の 2 次元データをクラスタリング

表 2: アプリケーションの実行環境

OpenCL デバイス	動作周波数 [GHz]	コア数
マルチコア CPU (Intel Core i7-6700)	3.40	物理コア 4 論理コア 8
dGPU (NVIDIA GeForce GTX 1060)	1.544	1,280
IGP (Intel HD Graphics 530)	0.35	24

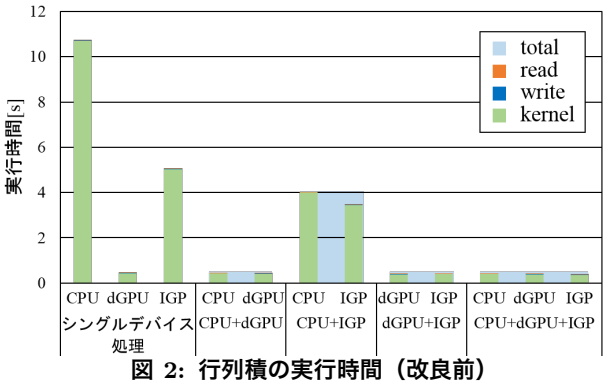

図 2: 行列積の実行時間（改良前）

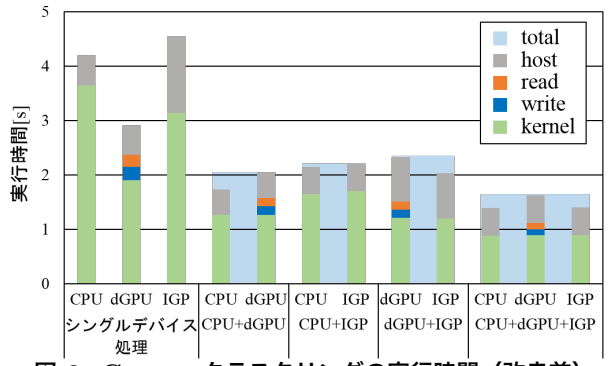

図 3: Canopy クラスタリングの実行時間（改良前）

3.3.2 実行結果と評価

改良前の手法での実行時間とその内訳を図 2 と図 3，改良後の手法での実行時間とその内訳を図 4 と図 5 に示す．これらのグラフにおける total はアプリケーション全体の実行時間，read は OpenCL デバイスからホストへのデータ転送時間，write はその逆方向のデータ転送時間，kernel はカーネル実行時間を表す．また，図 3 と図 5 における host は，Canopy クラスタリングに必要なホストでの処理時間を表す．なお，各実行時間は 10 回計測したうちで，total が最短となった際の内訳の時間である．

図 6 に，実行 OpenCL デバイスの各組合せにおける，改良前後の OpenCL デバイス使用率を比較したものを示す．ここで使用率とは，最も処理の遅い OpenCL デバイスの実行時間を 100%とした際の，最も処理の速い OpenCL デバイスの実行時間の割合のことである．使用率が高いほど OpenCL デバイスがアイドル状態となる時間が短く，マルチデバイス処理の効率が高いことを表す．改良前の手法では使用率が 85%を下回る OpenCL デバイスが生じるのに対し，改良後の手法ではすべて 94%以上となっている．

使用率の向上は実行性能の向上をもたらしている．例えば，行列積を CPU+IGP で実行した場合に，改良前の手法と比べて 10.7%の性能向上となった．

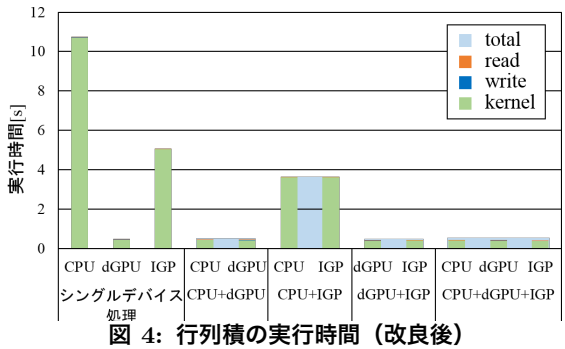

図 4: 行列積の実行時間（改良後）

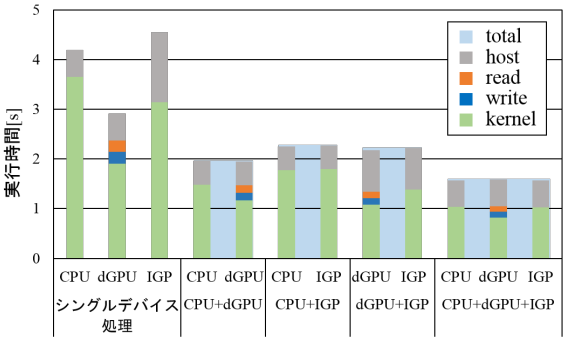

図 5: Canopy クラスタリングの実行時間（改良後）

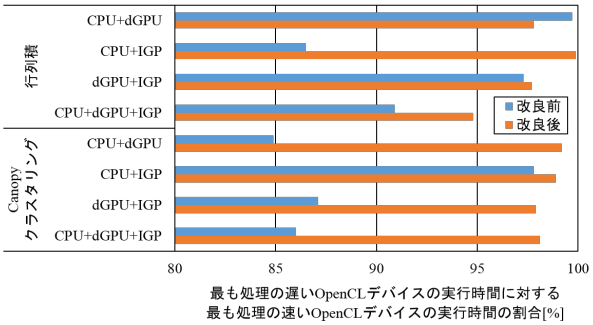

図 6: 改良前後の OpenCL デバイス使用率の比較

4. 提案する並列プログラミング手法

4.1 概要

3. で述べたように，改良後の負荷分散手法では，実行 OpenCL デバイスの組合せのすべてにおいて高い効率でマルチデバイス処理を行うことができた．本稿で提案する並列プログラミング手法はそのアプローチをもとに考案したものであり，効率的なマルチデバイス処理を行う OpenCL コードを自動生成するための手法である．

図 7 に，提案手法によって OpenCL コードを自動生成するまでの流れを示す．本手法に沿って OpenCL プログラミングを行う場合，設計者は C 言語ベースの言語記述で書かれたソースコードを用意するだけでよい．そのソースコードを入力として，変換系は並列化する処理とそうでない処理の識別と，並列化する処理で使用される変数と並列度の解析を行う．そして，効率的なマルチデバイス処理を行う OpenCL コードを生成する．

4.2 入力言語記述と出力 OpenCL コード

ここでは，入力として想定する言語記述と変換系が出力する OpenCL コードの詳細について，ベクトルの加算を行うプログラムを例に用いて述べる．

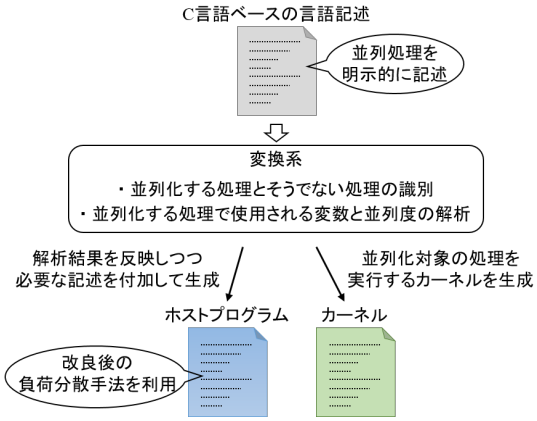

図 7: 提案手法による OpenCL コード自動生成

```
 5 int main() {
 6
 7   int vec_a[VEC_SIZE];
 8   int vec_b[VEC_SIZE];
 9   int result[VEC_SIZE];
10   int i;
          :
15   par_for (i = 0; i < VEC_SIZE; i++)
16     result[i] = vec_a[i] + vec_b[i];
17
18   return 0;
19 }
```

図 8: 入力として想定する言語記述の例

4.2.1 入力として想定する言語記述

図 8 に，本手法で入力として想定する言語記述の例を示す．15 行目の `par_for` 文が，C 言語に追加する並列処理用の構文である．文法そのものは C 言語の for 文と同じだが，これによって設計者は明示的に並列処理の記述を行うことができる．なお，`par_for` 文の本体で記述された処理は出力 OpenCL コードのカーネル内に変換される．

4.2.2 変換系が出力する OpenCL コード

変換系が出力する OpenCL コードは，カーネルとホストプログラムの 2 つである．ホストプログラムでは改良後の負荷分散手法を利用するため，処理の流れは次のようになる．

1) 実行 OpenCL デバイスのリストの作成
2) 各 OpenCL デバイスの実行性能のプロファイリング
3) 各 OpenCL デバイスに分散する負荷の算出
4) マルチデバイス処理の実行

これらの処理の大部分は，実行するアプリケーションの処理内容に依存せず同様のコードで記述可能なため，事前にコード片を用意してホストプログラムに付加する形となる．カーネルで使用する変数についての情報やカーネルの並列度などのアプリケーション依存の部分については変換系の解析結果が反映される．

以降では，カーネルについての詳細と，ホストプログラムのプロファイリング部分と分散負荷算出部分についての詳細を述べる．なお，本稿の執筆時点では変換系の実装は未着手のため，図 9〜図 11 に示す出力 OpenCL コードは人手によるものである．

カーネル

図 9 に出力されるカーネルを示す．まずカーネル関数の引数には，入力の `par_for` 文の本体（図 8 の例では 16 行目にあたる）で使用される変数が設定される．そして，起動されるワークアイテムの各インデックスは変数 `gidx` に格納され，

```
1  kernel void Kernel0(global int *vec_a,
2                      global int *vec_b,
3                      global int *result) {
4
5    int gidx = get_global_id(0);
6
7    result[gidx] = vec_a[gidx] + vec_b[gidx];
8
9  }
```

図 9: カーネルコード

```
146 for (i = 0; i < device_count; i++)
147   clEnqueueNDRangeKernel(
        queue[i], kernel[i], 1, NULL,
        &num_global_item_prof,
        &num_local_item, 0, NULL,
        &ev_kernel_prof[i]);
            ⋮
154 for (i = 0; i < device_count; i++){
155   clGetEventProfilingInfo(
        ev_kernel_prof[i],
        CL_PROFILING_COMMAND_START,
        sizeof(cl_ulong),
        &kernel_start_prof[i], NULL);
156   clGetEventProfilingInfo(
        ev_kernel_prof[i],
        CL_PROFILING_COMMAND_END,
        sizeof(cl_ulong),
        &kernel_end_prof[i], NULL);
157 }
158 for (i = 0; i < device_count; i++)
159   kernel_exec_time[i] = kernel_end_prof -
                            kernel_start_prof;
```

図 10: ホストプログラム（プロファイリング部分）

par_for 文の本体の処理は，ループ変数をワークアイテムのインデックス変数に置き換えてカーネルに出力される（図 8 の 16 行目から図 9 の 7 行目への変換）．

ホストプログラム（プロファイリング部分）

図 10 に，出力されるホストプログラムのうち，分散負荷の算出のために OpenCL デバイスの実行時間を測定する部分のコードを示す．まず，147 行目でプロファイリング用のカーネルを実行している．このとき，カーネル実行用 API の第 5 引数には変数 num_global_item_prof へのポインタが与えられ，この変数には起動するワークアイテムの数として，本来の処理に必要なワークアイテムの数に比べて十分小さな値が格納される．また，第 9 引数に OpenCL で定義されているイベントオブジェクト変数へのポインタを指定することで，カーネルの実行をイベントとして識別し，後の処理でそのイベントの実行時間を取得できるようにしている．そして 155 行目および 156 行目でプロファイリング用 API を実行し，159 行目で各 OpenCL デバイスのカーネル実行時間を取得している．

なお，図 10 には示していないが，データ転送にかかる時間についても同様の手段で取得可能である．

ホストプログラム（分散負荷の算出部分）

図 11 に，マルチデバイス処理のために各 OpenCL デバイスに分散する負荷を算出する部分のコードを示す．まず 170 行目から 175 行目までで，3.1 で示した式 (1) に相当する処理を行い，負荷分散比 distr_task_ratio を算出している．変数 exec_time_prof には，プロファイリング用カーネルの実行時間とそれに必要なデータ転送時間の和が格納される．そして 176 行目から 180 行目までで，distr_task_ratio の値を用いて，各 OpenCL デバイスに割り当てるデータ量 distr_data_amount

```
169 double tmp;
170 for (i = 0; i < device_count; i++){
171   tmp = 0;
172   for (j = 0; j < device_count; j++)
173     tmp += exec_time_prof[i] /
               exec_time_prof[j];
174   distr_task_ratio[i] = 1 / tmp;
175 }
176 distr_data_amount[device_count-1] = VEC_SIZE;
177 for (i = 0; i < device_count; i++){
178   distr_data_amount[i] =
       VEC_SIZE * distr_task_ratio[i];
179   distr_data_amount[device_count-1] -=
       distr_data_amount[i];
180 }
```

図 11: ホストプログラム（負荷分散の算出部分）

を分散負荷として算出している．マルチデバイス処理の実行時には，distr_data_amount の値をもとに各 OpenCL デバイスへのデータ転送や起動するワークアイテムの数の指定を行う．

5. 今後の課題

4. で述べた提案手法で用いる変換系が未実装のため，その実装が今後の課題となる．そして，実装した変換系によって出力される OpenCL コードの動作検証や実行性能の評価を行う必要がある．

また，GPU の性能を引き出すには GPU 特有のメモリを適切に使用することが求められるが，本稿の変換系の入力言語記述ではそれを明示的に行える記述を想定しておらず，C 言語の記述から自動的に使用メモリの指定を行うことも難しいと考える．そのため，GPU 特有のメモリの使用を明示的に指定する指定子等を，変換系の入力言語記述として新たに検討することも必要である．

6. おわりに

本稿では，異種プロセッサ環境における並列プログラミング手法として，並列処理を明示的に記述可能な言語記述から，与えられる環境において最適な並列処理を実現する OpenCL コードを自動生成する手法を提案した．まず異種プロセッサの並列動作のための負荷分散手法について述べ，その後，自動生成に必要な入力言語記述と生成される OpenCL コードの詳細について，単純なアプリケーションを例に用いて述べた．入力言語記述から OpenCL コードを出力する変換系については現在未実装のため，今後はその実装と出力 OpenCL コードの動作検証や実行性能の評価，その他の改良を行う予定である．

参考文献

[1] OpenCL Overview - The Khronos Group Inc, https://www.khronos.org/opencl/.

[2] Marvin Damschen, Frank Mueller, Jorg Henkel, “Co-Scheduling on Fused CPU-GPU Architectures With Shared Last Level Caches,” IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, Vol. 37, No. 11, pp. 2337−2347, Nov. 2018.

[3] Qualcomm Technologies, Inc., *Qualcomm Snapdragon Mobile Platform OpenCL General Programming and Optimization*, Nov. 2017.

[4] 林 卓哉, 渡邊 誠也, 名古屋 彰, “OpenCL を用いたマルチデバイス処理向けのタスク振り分け手法,” 平成 30 年度（第 69 回）電気・情報関連学会中国支部連合大会 講演論文集, R18-19-06, Oct. 20, 2018.