Vol.2019-SLDM-189 No.23
Vol.2019-EMB52 No.23
2019/11/14

マルチコア向け OpenCL フレームワークの Raspberry Pi 上での性能評価

宮崎貴史[1]　左隼人[1]　北條直久[1]　谷口一徹[2]　冨山宏之[1]

概要：現在までに我々は，マルチコア・アーキテクチャを対象とした OpenCL フレームワークを開発してきた．本論文では，本 OpenCL フレームワークを教育用コンピュータ Raspberry Pi 上に移植し，性能評価を行った結果を報告する．

キーワード：組込みシステム，OpenCL，マルチスレッド，マルチコア，データ並列

1. はじめに

OpenCL[1]は並列コンピューティングのための標準化されたフレームワークのひとつである．OpenCL はそのオープンな標準であることやハードウェアプラットフォームに依存しないことから既に広く普及している．GPU，マルチコアプロセッサ，および FPGA といった様々なハードウェアプラットフォーム上で実行可能な OpenCL プログラムが多く存在しており，プラットフォーム間での移植が容易である．しかし，構成と特性の違いから，移植に際して処理効率が維持される保証はない．GPU と汎用マルチコアプロセッサとでは並列性の粒度が異なるため，GPU 向けに記述されたプログラムの多くは汎用マルチコアプロセッサ上では処理速度性能が低下する．本研究では，GPU 向け OpenCL プログラムを Raspberry Pi[2]上のマルチコアプロセッサで効率的に実行する方法について扱う．

GPU 向けの OpenCL プログラムをマルチコアプロセッサ上に移植する試みはいくつか既に行われている．文献[3]では Dong らが様々なハードウェアプラットフォーム上で効率的に実行できるような統一的なプログラミングスタイルの定義を試みている．文献[4]では Shen らが GPU 向け OpenCL プログラムをマルチコアプロセッサ上への移植について取り組んでいる．文献[4]では，結論のひとつとして，プログラマは系統立った方法によって最適な並列性の粒度を見つけ出さなければならないと述べられている．文献[5]では Seo らが，多面体モデルに基づいた OpenCL プログラムのワークサイズの選択アルゴリズムを提案している．文献[6]では Miyazaki らがマルチコアプロセッサ上で多数のスレッドを効率的に処理する方法を提案し，内製の OpenCL フレームワークである RuCL に実装し評価している．

システム設計において性能解析を行い，性能ボトルネックを発見することが重要である．近年では性能解析のためのツールが多数開発および公開されており，Linux における perf もそのひとつとして有名である．perf は Linux のカーネルに含まれる複数の性能解析ツールをまとめたものであり，Linux システムあるいはシステム中の特定のプログラムが動作中のカーネル領域およびユーザ領域の各種イベントを観測できる．観測できるイベントの中には CPU のパフォーマンスモニタリングカウンタ (PMC) から取得できる値も含まれている．ページフォールトなどのカーネルカウンタから取得できるイベントはソフトウェアイベント，実行命令数などの CPU の PMC から取得できるイベントはハードウェアイベントと呼ばれる．

本論文では組込みマルチコア向け OpenCL フレームワークを教育用ボードコンピュータ Raspberry Pi 3B+上で動作させた際の性能を評価する．本研究は文献[6]にて提案されている RuCL フレームワークを直接の先行研究としている．RuCL フレームワークは組込みマルチコア上で GPU 向け OpenCL プログラムを効率的に動作させるための OpenCL フレームワークである．文献[6]における性能評価の環境は，サーバ向けメニーコアプロセッサおよび数十 GB のメモリを搭載したワークステーションであったため組込みシステム向けフレームワークを評価するのに適していなかった．本研究では，組込みマルチコアプロセッサを搭載している Raspberry Pi 3 B+[2]を性能評価の環境とした．Raspberry Pi は 4 コアプロセッサおよび 1GB メモリを搭載するボードコンピュータである．本来 Raspberry Pi は教育目的のコンピュータであるが，その高い使用性と機能性から組込みコンピュータとして採用されている実績が多数ある．したがって，組込み向けマルチコア向け OpenCL フレームワークを評価するのに適当である．また，本研究では perf を用いてより詳細に性能の解析を行った．文献[6]ではマルチスレッドプログラムにおけるスレッド制御方式を複数提案しており，制御方式ごとに性能差が表れていたが，その原因については推測するに留まっていた．本研究では Linux カーネルに含まれる性能解析ツール perf を用いてソフトウェアイベントおよびハードウェアイベントを観測し，スレッド制御方式によって各性能指標がどのような値を取るかを比較し検討する．

本論文の構成は以下の通りである．第 2 章では RuCL フレームワークおよび OpenCL の概要について述べる．第 3 章で RuCL の Raspberry Pi 上での評価実験について述べる．

1 立命館大学
2 大阪大学

ⓒ2019 Information Processing Society of Japan

Vol.2019-SLDM-189 No.23
Vol.2019-EMB52 No.23
2019/11/14

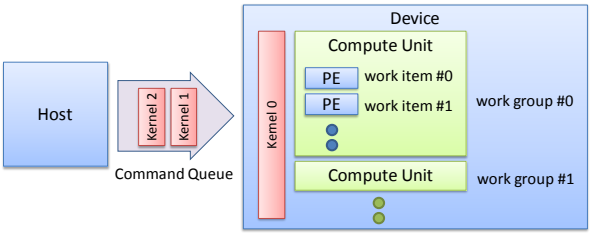


図 1　OpenCL におけるデータ並列

最後にまとめを行う．

2. RuCL フレームワーク

本章では，RuCL フレームワークについて述べる [6]．まず OpenCL の概要について述べる．次に RuCL フレームワークの概要およびスレッド実行方式について述べる．最後に文献[6]にて行ったワークステーション上での実験結果について紹介する．

2.1 OpenCL の概要

OpenCL は並列コンピューティングのための標準化されたフレームワークのひとつである．OpenCL はデータ並列実行とタスク並列実行をサポートしているが，本研究ではデータ並列実行にのみ注目する．図 1 は OpenCL におけるデータ並列実行のモデルを示したものである．この図において，計算機ハードウェアはひとつのホストプロセッサとひとつのデバイスから構成される．デバイスは複数のコンピュートユニット (CU) から構成され，ひとつの CU は複数のプロセッシングエレメント (PE) から構成される．デバイスにより執行されるプログラムのコードはカーネルと呼ばれる．実行カーネルはホストからコマンドキューを介してデバイスに対して割り当てられる．データ並列実行の場合，同じカーネルがデバイス中の複数の CU で動くことになる．データはワークグループの組に分割され，CU に割り当てられる．ひとつのワークグループは複数のワークアイテムを持ち，各ワークアイテムは CU 中の PE で処理される．本質的にワークアイテムはスレッドに対応している．バリア同期は同じワークグループに所属するワークアイテムの間でだけ可能である．

2.2 RuCL フレームワークの概要

RuCL は OpenCL フレームワークの一つで，主に組込みシステム向けのマルチコアプロセッサを対象としている [6]．特にオブジェクトの作成とワークアイテムとスレッドの制御が組込みシステム向けに設計されている．実行時のオーバーヘッドを削減するため，コンテキストやコマンドキューなどオブジェクトの作成を静的に行うように設計されている．どのようなオブジェクトを生成するかはプログラムのコンパイル時点で決定される．RuCL フレームワークのライブラリは内部で POSIX threads (pthreads) を用いている．OpenCL プログラムによって記述されたワークアイテムを処理するスレッドを POSIX スレッドを用いて生成し，データ並列処理を行う．基本的に OpenCL フレームワークは OpenCL プログラムによって指定されたワークアイテム数に対応した数のスレッドを生成する．しかし，RuCL においては，ワークアイテム数はスレッド数と一対一対応するとは限らない．RuCL フレームワークにおいてはワークアイテムとスレッドの対応関係がプログラマが任意に指定できるスレッド実方式によって決定される．RuCL のスレッド実行方式は GPU 向け OpenCL プログラムをマルチコアプロセッサ上で効率的に動作させるために重要な部分である．現代的な GPU は数千にも及ぶ多数の GPU コアを持ち，高速なスレッド切り替えのハードウェアを備えている．そのような GPU は数十億にも及ぶ巨大な数のスレッドを同時に，かつ，効率的に処理することができる．それゆえに，多くの GPU 向け OpenCL プログラムは多数の小さいスレッドを生成し処理するように作成されている．一方，汎用マルチコアプロセッサは GPU ほど多数のプロセッサコアは持たない．例えば，典型的なデスクトップ PC 向けマルチコアプロセッサのコア数は 4 から 32 の間である．さらに，マルチコアプロセッサにおけるスレッド切り替えはソフトウェアに依存しており，GPU に比べるとその速度は極

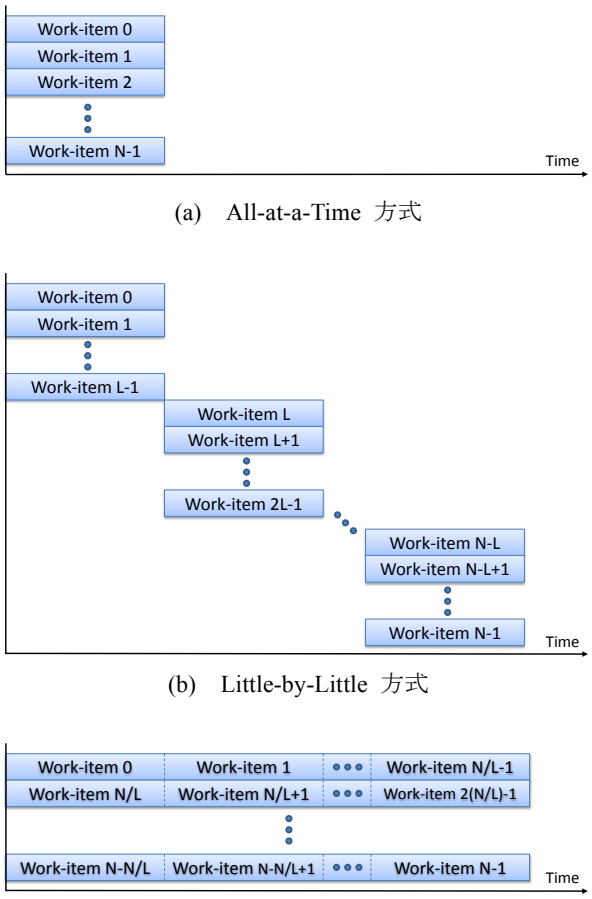


図 2　RuCL フレームワークのスレッド実行方式

ⓒ2019 Information Processing Society of Japan

Vol.2019-SLDM-189 No.23
Vol.2019-EMB52 No.23
2019/11/14

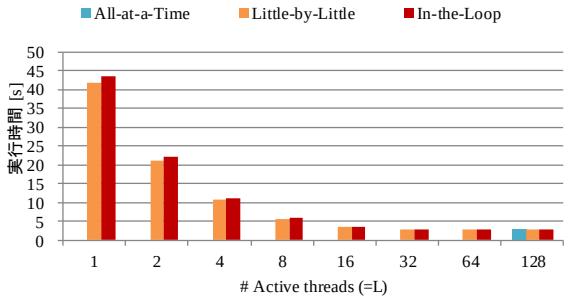
めて遅い．したがって，マルチコアプロセッサが同時に巨大な数のスレッドを扱うことは困難である．これが GPU 向け OpenCL プログラムがマルチコアプロセッサ上で効率的に動作できない主な理由である．GPU 向けに記述された OpenCL プログラムは，マルチコアプロセッサ上で実行できるものの，その実行速度は多くの場合期待されるほど高速ではなくなる．多数のワークアイテムをマルチコアプロセッサ上で実行可能な数のスレッドへと対応させる方法が RuCL のスレッド実行方式として提案されている．

2.3 スレッド実行方式

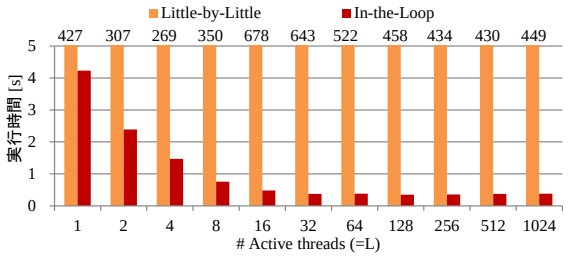
文献[6]では，著者らはマルチコアプロセッサ上での OpenCL フレームワーク RuCL，および RuCL に含まれる 3 つのスレッド実行方式を提案，実装および評価を行っている．文献[6]にて提案されているスレッド実行方式は(a) All-at-a-Time 方式，(b) Little-by-Little 方式，および(c) In-the-Loop 方式と名付けられており，3 つのスレッド実行方式の概要を図 2 に示す．図の横軸はプログラム実行中の経過時間を表し，縦軸は並列に動作するスレッドの個数を示している．図中の N は OpenCL プログラムで指定されるワークアイテムの数を意味している．All-at-a-Time 方式は最も簡潔な実装であり，物理的な PE の数とは関係なく N 個のスレッドが同時に生成され，処理される．この実行方式ではワークアイテムとスレッドは一対一対応し，N が小さい数の場合はうまく動作する．しかしながら，この実行方式ではオペレーティングシステムが扱いきれないほど N が大きくなった場合，マルチコアプロセッサ上での実行はできなくなってしまう．この問題の対策として Little-by-Little 方式と In-the-Loop 方式が提案された．Little-by-Little 方式と In-the-Loop 方式はワークアイテムとスレッドが一対一対応しない．Little-by-Little 方式は少数ずつスレッドを生成し，処理を行うことを繰り返す．ここで同時に生成し処理できるスレッドの最大の数をアクティブスレッド数 L とする．この L は OpenCL プログラムに含まれるワークアイテム数を超えない範囲でプログラマが任意に決定できる．Little-by-Little 方式ではスレッドの生成および終了が各 N/L 回行われることになる．In-the-Loop 方式は最初に L 個のスレッドを生成し，各スレッドは N/L 個のワークアイテムを繰り返し処理する．In-the-Loop 方式ではスレッドの生成および終了は L 回行われることになる．

2.4 RuCL のワークステーション上での評価

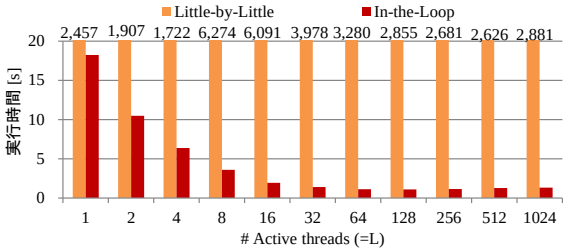
図 3 は文献[6]で示されている RuCL フレームワークのワークステーション上での実験結果を再編集したものである．図 3 (a)，(b)，および (c) はベンチマークプログラム集 BEMAP[7]に含まれるベンチマークプログラムである Montecarlo，Black-Scholes，および Linear-Search を 3 つの実行方式で動作させた実行時間をグラフにしたものである．ここで示されている In-the-Loop 方式は文献[6]において改良されたものである．実験に使用したワークステーション



(a) Montecarlo　(128 ワークアイテム)



(b) Black-Scholes　(10,485,760 ワークアイテム)



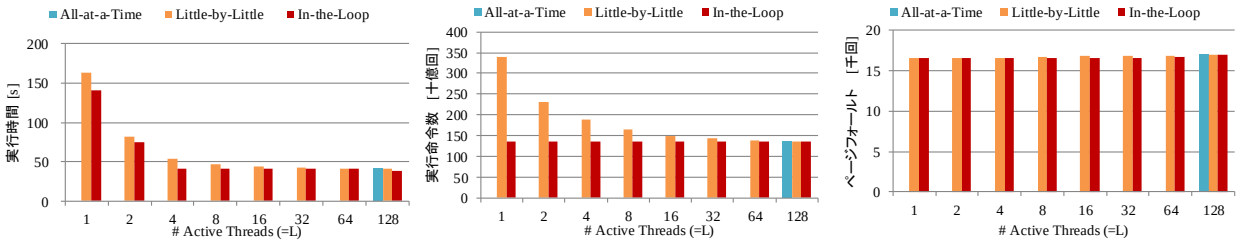
(c) Linear-Search　(67,108,864 ワークアイテム)

の CPU はデュアル Xeon E5-2620 (物理コア 12 個，論理コア 24 個) であり，メモリは DDR3RAM 64GB である．

図 3 (a) は Montecarlo プログラムを使用した 3 種のスレッド実行方式の比較である．Montecarlo プログラムが含むワークアイテムは 128 個のみであるので，All-at-a-Time 方式が実行時間最短となり，十分良い結果を示した．また，Little-by-Little 方式, In-the-Loop 方式はホストコンピュータの論理コアの数よりもアクティブスレッドの数が小さい場合はあきらかに All-at-a-Time 方式に比べ劣る結果となった．

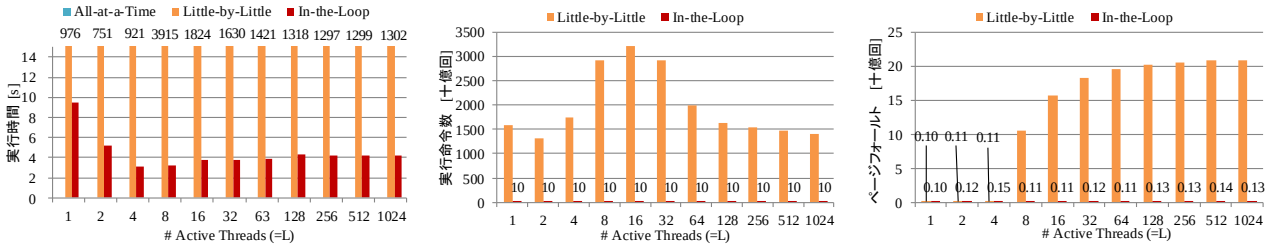
図 3 (b) は Black-Scholes プログラムの実行結果を示したものである．ワークアイテムの数が千万を超えるため，All-at-a-Time 方式はスレッドの生成ができず実行に失敗した．Little-by-Little 方式は実行できているが，膨大な数のスレッドの生成と終了を繰り返すオーバヘッドのため効率的でない．In-the-Loop 方式はアクティブスレッド 128 個の場合が実行時間が最短であり，349 ミリ秒であった．Linear-Search の実験結果は図 3 (c) のようになっている．Black-Scholes と同様に Little-by-Little 方式が効率的でなかった．実行時間が最短となったのはアクティブスレッド

ⓒ2019 Information Processing Society of Japan



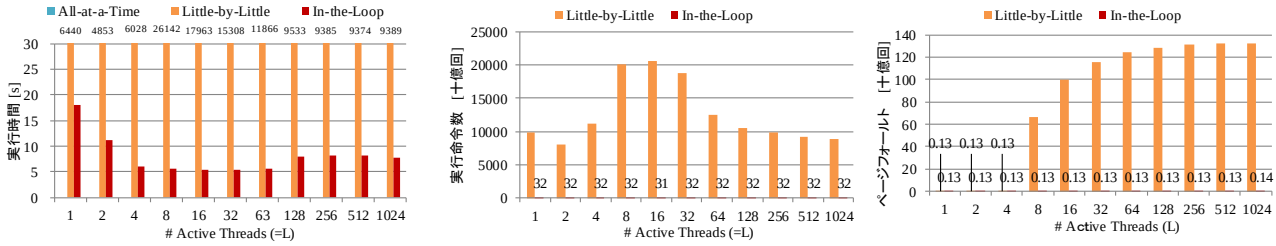
(a) 実行時間　(b) 実行命令数　(c) ページフォールト数

図 4　Montecarlo の実験結果 (128 ワークアイテム)



(a) 実行時間　(b) 実行命令数　(c) ページフォールト数

図 5　Black-Scholes の実験結果 (10,485,760 ワークアイテム)



(a) 実行時間　(b) 実行命令数　(c) ページフォールト数

図 6　Linear-Search の実験結果 (67,108,864 ワークアイテム)

128 個の In-the-Loop 方式であり，1089 ミリ秒であった．

評価実験はスレッド実行方式ごとの傾向を評価し，In-the-Loop 方式が多数のスレッドを処理するのに有効であることが分かった．しかしながら，RuCL フレームワークは組込みマルチコア CPU 上で動作させることを想定したものであり，ワークステーションは本来使用が想定される環境ではないため，十分な評価とはいえなかった．

3. RuCL の Raspberry Pi 上での評価

RuCL フレームワーク[6]の性能を実験環境を Raspberry Pi 3 B+として評価した．性能の評価指標として，実行時間および Linux perf によるソフトウェアイベントおよびハードウェアイベントの起こった回数の測定を行った．

3.1 実験の環境と方法

実験の環境として Raspberry Pi 3B+ (ARM Cortex-A53 4 コア，メモリ 1GB)を使用した．Raspberry Pi 上では Ubuntu Server 18.04 LTS が OS として動作している．コンパイラは文献[6]の実験と同様に g++4.8 とした．ベンチマークプログラムとして，BEMAP[7] から 3 つのプログラム Montecarlo，Black-Scholes，Linear-Search を使用した．これらのプログラムは文献[6]で使用されたものと同様のものである． All-at-a-Time，Little-by-Little，In-the-Loop の 3 種のスレッド実行方式でベンチマークプログラムを動作させたときの実行時間，実行命令数，およびページフォールト数を比較した．実行命令数とページフォールトの観測には perf(4.15)のサブコマンドである stat を用いた．実験に用いた In-the-Loop 方式は文献[6]における改良された In-the-Loop 方式である．

3.2 実験結果と考察

図 4 に Montecarlo の実験結果を示す．図 4 (a) に示す通り，実行時間が最短となったのは L=128 のときの In-the-Loop 方式である．Little-by-Little 方式および In-the-Loop 方式ともにアクティブスレッド数が増加するにつれ実行時間は減少している．アクティブスレッド数が同

ⓒ2019 Information Processing Society of Japan

じとき，In-the-Loop 方式が Little-by-Little 方式より実行時間が短くなるという，ワークステーションでの実験とは異なる結果となっている．図 4 (b) から，アクティブスレッド数が同じとき，In-the-Loop 方式が Little-by-Little 方式より実行命令数が少ないことが分かる．図 4 (c) から，ページフォールトに関してはスレッド実行方式ごとに目立った差異は見られない．これの原因として Montecarlo のワークアイテム数は 128 と小さいこと，および処理するデータサイズが小さいことなどが考えられる．

図 5 に Black-Scholes の実験結果を示す．図 5 (a) に示す通り，実行時間が最短となったのは L=4 のときの In-the-Loop 方式である．All-at-a-Time 方式は実行に失敗している．これは生成しようとするワークアイテム数と同数のスレッドが Linux のスレッド数上限を超えるためである．In-the-Loop 方式の実行時間は，L=4 まで減少し，以降増加し，L=128 で飽和するような形となっている．これはワークステーションとは異なる傾向である．図 5 (b) から，実行命令数に関しては In-the-Loop 方式はアクティブスレッド数に対して大きな変化はない．一方，アクティブスレッド数に対して Little-by-Little 方式の実行命令数の変化は大きく，実行時間の変化と似た変化をしている．図 5 (c) から，ページフォールトに関しては In-the-Loop 方式は大きな変化がないものの，Little-by-Little 方式はアクティブスレッド数 8 から急激にページフォールトが増加している．これは，Black-Scholes が持つワークアイテムが多いことと，ワークアイテムあたりのデータサイズが大きいため，スレッドの生成および終了を繰り返す Little-by-Little 方式におけるメモリオーバヘッドが大きくなったのだと考えられる．

図 6 に Linear-Search の実験結果を示す．図 6 (a) に示すと通り，実行時間が最短となったのは L=16 のときの In-the-Loop 方式であった．All-at-a-Time 方式は Black-Scholes の場合と同様に実行に失敗した．アクティブスレッド数に対する実行時間の変化は Little-by-Little 方式と In-the-Loop 方式ともに，Black-Scholes の結果と似た概形となった．図 6 (b) および (c)から，実行命令数およびページフォールト数も Black-Scholes で見られたものと同様の傾向が見られた．

以上の結果から，組込みマルチコアを搭載する Raspberry Pi 上での評価でも，過去のワークステーション上での評価と同等かそれ以上に，In-the-Loop 方式の有効であることが示せた．また， Raspberry Pi のようにメモリが極めて少ないシステム上で Little-by-Little 方式にて多数のワークアイテムを含むプログラムを動作させた際には，実行命令数やページフォールトが大量に発生し，実行時間に影響を与えることが示唆された．

4. おわりに

本論文では，マルチコア・アーキテクチャを対象とした OpenCL フレームワークを教育用コンピュータ Raspberry Pi 3B+上に移植し，性能評価を行った．Raspberry Pi 上では In-the-Loop 方式が実行時間最短であり，その有効性が示せた．また，過去の研究で行った実験結果と今回の実験結果を比較すると，同じプログラムでも実験環境によって最適な並列数は異なることが分かった．

現在提案している RuCL フレームワークでは，ワークアイテムはスレッドに対して静的に割り当てられるようなっている．今後の研究ではスレッドの割り当てを動的に行い，より良い負荷分散を行えるよう割り当てアルゴリズムを改善することを計画している．また，今回評価に用いた性能指標はかならずしも多くなかった．より多くの性能指標から多面的に性能を評価することが求められる．

謝辞 本研究は一部，キオクシア株式会社（旧社名 東芝メモリ株式会社）の支援による．

参考文献

[1] The OpenCL Specification, https://www.khronos.org/opencl/. (2019 年 10 月アクセス)
[2] Raspberry Pi Foundation, https://static.raspberrypi.org/files/product-briefs/Raspberry-Pi-Model-Bplus-Product-Brief.pdf. (2019 年 10 月アクセス)
[3] H. Dong, D. Ghosh, F. Zafar, and S. Zhou, “Cross-platform OpenCL code and performance portability investigated with a climate and weather physics model,” International Conference on Parallel Processing Workshops (ICPPW), pp. 126-134, 2012.
[4] J. Shen, J. Fang, H. Sips, and A. L. Varbanescu, “An application-centric evaluation of OpenCL on multi-core CPUs,” Parallel Computing, vol. 39, no. 12, pp. 834-850, 2013.
[5] S. Seo, J. Lee, G. Jo, and J. Lee, “Automatic OpenCL work-group size selection for multicore CPUs,” International Conference on Parallel Architectures and Compilation Techniques (PACT), pp.378-397, 2013.
[6] T. Miyazaki, H. Hidari, N. Hojo, I. Taniguchi and H. Tomiyama, "Revisiting thread execution methods for GPU-oriented OpenCL programs on multicore processors," International Workshop on Advances in Networking and Computing (WANC) in conjunction with International Symposium on Computing and Networking (CANDAR), pp. 520-523, 2018.
[7] Y. Ardila, N. Kawai, T. Nakamura, and Y. Tamura, “Support tools for porting legacy applications to multicore,” Asia and South Pacific Design Automation Conference (ASP-DAC), pp.568-573, 2013.

ⓒ2019 Information Processing Society of Japan