Vol.2011-ARC-196 No.12
2011/7/27

OpenCLの性能可搬性改善に向けた基本APIの提案

京　昭倫[†1]　岡崎　信一郎[†1]

アクセラレータ用標準並列言語として提案されているOpenCLは、各種並列言語や並列化フレームワークが乱立する現状を改善する可能性を持つ。しかしながら、GPUのメモリアーキテクチャを前提に仕様策定されたOpenCLそのままでは、GPUとは異なったメモリアーキテクチャを持つアクセラレータの演算能力を引き出すのは困難である。この問題点に対し、本稿ではOpenCLカーネルを直接ではなく、「自動データ転送」を行うシステム関数を介して呼び出すことにより、OpenCLカーネル内から、アクセラレータ間で性能差が出やすいグローバル空間アクセス記述を排除するためのプログラミングフレームワーク、並びにそのための基本API群を提案する。

1. はじめに

近年、デスクトップCPUの分野でも発熱を抑える必要性から、動作周波数向上ではなく、コア数を増やす方向で性能向上を実現する動きが顕著になってきているが、特定分野向けでは、以前よりマルチ・メニーコア化が盛んであり、様々な用途向けに、メニコア型のアクセラレータが提案されている[1)2)3)4)5)]。しかしながら、アクセラレータ毎に相異なる様々な並列言語や並列化フレームワークが開発されたことで、プログラムコード間で移植性がないという課題があった。そうした中、2009年頃にアクセラレータ用標準並列言語として提案されたOpenCL[6)]は、この状況を大きく改善する可能性を持つ。しかし、OpenCLは主にグラフィックス分野向けアクセラレータであるGPUの活用を前提に仕様策定されたため、GPUと異なる設計思想を持つ他のアクセラレータの上では、残念ながら開発されたOpenCLコードがそのままでは効率よく動作しないという問題点がある。

OpenCLに関する論文は2009年以降、OpenCLによる既存アプリケーションのGPUへの移植[7)8)20)]、OpenCLとCUDAの性能比較[18)19)20)]、OpenCLをCELL.B.Eに適用した場合の有効性評価[17)]、OpenCLを用いた異種GPUの利用効率最適化[9)10)]、そしてOpenCLと他の並列化フレームワーク(OpenMPとPthread)との有効性比較[12)]等、多く発表されている。中でも文献[12)]では、OpenCLの欠点として高い記述煩雑性と低い性能可搬性を挙げており、その対策としてreduction、scan, gather, scatter, split, そしてquicksort等の利用頻度の高い並列演算(parallel primitives)をAPIとして導入することを提案している。

本稿では、OpenCLカーネルコード(以降カーネル)の記述煩雑性と性能可搬性の改善に向け、異種アクセラレータ間でのメモリアーキテクチャの相違を吸収する仕組みの導入を提案する。これは、例えば文献[12)]とは互いに補間関係にあるアプローチとして位置づけられる。以下2章では、OpenCLが想定しているアクセラレータのアーキテクチャ構成について述べると共に、解決すべき課題を整理する。3章では、データ転送を自動化する提案フレームワークの詳細、並びにその利用に向けた基本API群を与える。4章では、異なるメモリアーキテクチャを持つ3種類のアクセラレータに対する提案フレームワークの有効性の予備評価を行う。最後に5章でまとめと今後の予定について述べる。

2. OpenCLアーキテクチャの特徴と課題

2.1 OpenCLアーキテクチャ

OpenCLアプリケーションは、ホスト側で動作するホストコードと、OpenCLデバイス(=アクセラレータ)側で動作するカーネルとで構成される。OpenCLデバイスに演算処理を行わせるには、1) 参照データやカーネルコードをホスト側からOpenCLデバイス側へ転送する、2)「カーネル起動コマンド」でOpenCLデバイス上でのカーネル実行を開始させる、そして3) カーネルの実行完了後、OpenCLデバイスのメモリ空間から結果データをホスト側に読み出す、の順でOpenCLホストAPIをコールする。

OpenCLデバイスの「演算部」は一つ以上の演算ユニット(CU:Compute Unit)からなり、演算ユニットは一つ以上の演算要素(PE:Processing Element)からなる、というように階層構造をなす(図1(a))。OpenCLデバイスの「記憶部」も同様に、全PEからアクセス可能なグローバル(Global)メモリとコンスタント(Constant)メモリ、同一CUに属するPE群のみがアクセス可能なローカル(Local)メモリ、そして個々のPEのみがアクセス

†1 ルネサスエレクトロニクス株式会社 技術開発本部 先行研究統括部
Advanced LSI Systems Research, LSI Research Laboratory, Renesas Electronics Corporation

可能なプライベート (Private) メモリからなる。処理のマッピングは図 1(b) に示すような実行レイアウト、すなわち利用 work-item 数 (Global work-item size) とその次元数[*1]の指定により行う。その際、さらに work-item 群をグループ (work-group) に分けるかどうか、また分ける場合はそのサイズ[*2]を指定する。

全 work-item が SIMD 制御の下、同一のカーネルを実行する SIMD モード以外に、個々の work-item が別々のカーネルを実行する MIMD モードも存在するが、本稿では SIMD モード実行のみを対象とする。

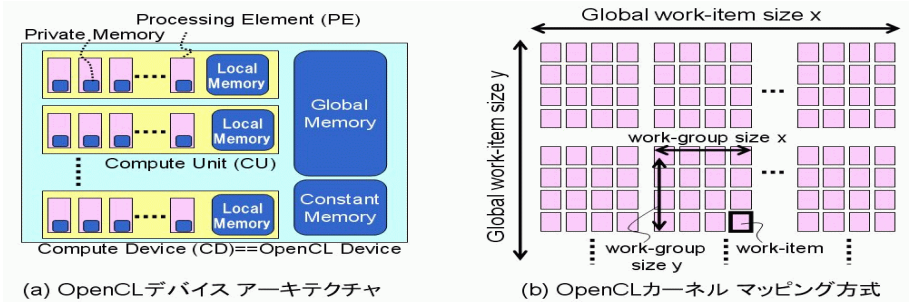

図 1 OpenCL デバイスのアーキテクチャ[6)]

2.2 課題の整理

逐次プロセッサでは多くの場合、記憶部が物理的にはオンチップの小容量 Private メモリとオフチップの大容量 Global メモリとに分かれていても、キャッシュ制御機構の導入で一つの大きなメモリ空間としてユーザに見せることで、プログラム開発の容易化が計られている。しかし並列プロセッサ、特に多数のコアを搭載したメニコア型アクセラレータとなると、コア (=PE) 数だけ Private メモリが存在するようになり、それらを全て一つのキャッシュ制御機構で統一的に管理するのは、ハードコストが高く一般に実現困難である。前述の OpenCL デバイスで想定された階層型メモリアーキテクチャも、そうしたことを考慮した上での一つの現実解といえる。しかしながらその結果、ユーザからは複数のメモリ空間が見えるようになり、性能追求に際しては、メモリ空間同士の速度差、容量差、機能差等に関する知識を持つことが必要となる (図 2)。

OpenCL の場合、言語仕様上では、役割の異なる 4 種類のメモリ空間が階層構造内に点在する。その結果、カーネルの性能を充分に引き出そうとするユーザは一般的には、利用頻度の高いデータを、なるべくより高速なメモリ空間に移動してから参照しようとする。しかし、同じ OpenCL 対応のデバイスでもメーカーが異なれば、各メモリ空間の容量、アクセス速度、アクセス遅延、そしてキャッシュ制御の有無、等の点で必ず差異が存在する。そのため、ある OpenCL デバイスにとり理想的な階層間のデータ移動でも、他の OpenCL デバイス、あるいは世代が変われば同一シリーズの OpenCL デバイスにとっても、かえって性能を悪化してしまう原因となる可能性がある。また、そうした性能チューニング用コードの設計は、アルゴリズム設計者からみれば本質的でないため負担に感じる作業である上に、コードの可読性低下、コード煩雑度増大等をも引き起こす。

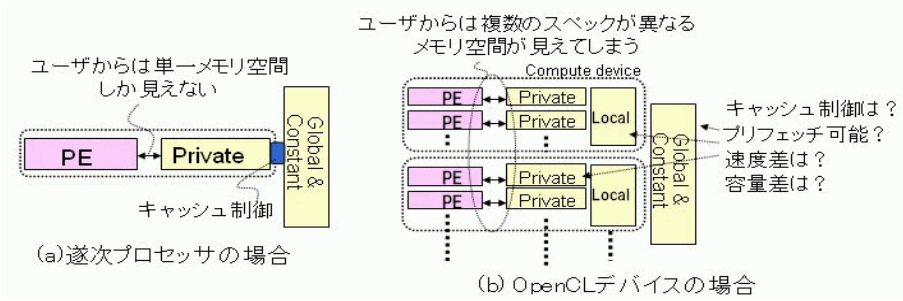

図 2 逐次プロセッサとアクセラレータ間でのユーザ見えメモリ空間の相違

上記の諸問題が発生するのは、メモリアーキテクチャに対するデバイス間の設計相違を隠蔽する仕組みが、現状 OpenCL で用意されていないことが主な原因と考えられる。逐次プロセッサの標準言語である C 言語では、演算に本質的な部分以外の、例えば I/O 部分のようにプロセッサ間で設計相違が出やすい部分は、標準ライブラリ (API) の形で詳細を隠蔽する方法が取られてきた。OpenCL が今後、標準的な並列言語プラットフォームとして、GPU に限らず他のアクセラレータにも普及していくためには、少なくとも C 言語と同レベルの可読性・性能可搬性を持つことが重要である。それには今後、OpenCL デバイス間で設計相違が出やすい部分であるメモリアーキテクチャの詳細を、効果的に隠蔽する戦略がより一層求められる。

⋆1 最大 3 次元まで、図 1(b) では 2 次元の場合を例示
⋆2 一つの Work-group 内に存在する work-item 数

3. ACL フレームワークの提案

異種 OpenCL デバイス間でのメモリアーキテクチャの設計相違の隠蔽、そしてユーザカーネルの可読性向上と設計容易化を目的に、Global 空間と Private/Local 空間の間で自動的にデータ転送を行うライブラリ (Automated Communication Library、以降 ACL と呼ぶ) の開発、そしてその利用を前提にユーザカーネルの設計を行う枠組みを提案する。以降、こうした枠組みを「ACL フレームワーク」と呼ぶ (図 3)。

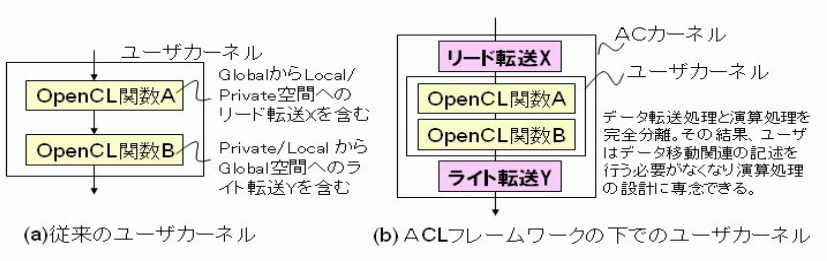

図 3 ACL フレームワークの概要

ACL フレームワークでは、OpenCL ホストとカーネルの間に、データ転送を自動的に行う「AC カーネル」層をさし挟む。ユーザはホスト側の処理として、カーネル内で参照または定義したいデータ群のそれぞれに対し、転送パタンを指定した上で、AC カーネルを起動する。AC カーネルでは、指定された転送パタン群を元に、転送シナリオを決定した後、それに基づき、データ転送並びにユーザ処理の起動を、必要回数だけ繰り返して行う。そしてユーザ処理内では、前記の転送シナリオに従って、Global 空間内の前記データ群が、Private/Local 空間へ自動的に転送されるものと想定し、AC カーネルから渡される Private/Local 空間へのポインタを利用するよう処理記述を行う。

こうした枠組みにより、ユーザは性能を引き出すための煩雑なデータ転送動作の設計から解放され、かつデータ転送作業がターゲット毎に対する熟練者によって設計される AC カーネルに委ねられるため、ユーザ処理コードの性能可搬性を大きく改善する効果が期待される。

ACL フレームワークが有効に機能するためには、ユーザの所望するデータ転送処理を多くの場合、転送パタンの指定そして転送シナリオの作成によって再現できなければならない。これには、ユーザコードが何らかの規則性をもった Global 空間アクセスを行っている必要があるが、既に並列化で性能向上の恩恵を蒙っているアプリケーションの多くが、規則性的なメモリアクセスパタンを持つ[14)13)18)20)] ことを考えると、大きな制約にはならない。

3.1 転送対象データ群の指定方法

ACL フレームワークでは、従来の OpenCL カーネルが持つ、Global 空間へのポインタ引数のうち、ユーザが指定したものが指すアドレスを先頭するデータ群を、AC カーネルが転送対象データ群と見なす。

ユーザはまず、個々の転送対象データ群に対し、次に述べる転送属性を指定することを通じ、転送パタンを設定する。次に、こうして指定された複数の転送パタンを元に、AC カーネルが転送シナリオを決定する。そして転送対象データは、転送シナリオに従って、必要に応じて分割された形で、Global 空間と work-item 毎の Private/Local 空間との間で自動的に転送される。

3.2 転送属性に基づく転送パタンの指定

転送対象に指定された個々のポインタ引数に対する転送パタンは、ユーザが、ホスト側でのカーネル起動の準備としてカーネル引数を設定する際に、下記に示す転送属性群を指定することにより行う。

(1) **サイズ属性**：最大で 3 次元まで、各次元のデータサイズをワード数で指定することで、転送対象データ群全体のワード数を指定する。なおワード毎のバイト数も合わせて指定する。

(2) **参照属性**：リード方向 (Global 空間から Private/Local 空間へ) の転送が不要ならば $NONE$、そうでなければ転送動作の開始位置を TOP_LEFT、$BOTTOM\_LEFT$、そして $RANDOM$ のいずれかで指定する。$RANDOM$ が指定された場合に限り、別途、開始位置の情報を格納した領域へのポインタを指定する。

(3) **定義属性**：ライト方向 (Private/Local 空間から Global 空間へ) の転送が不要ならば $NONE$、そうでなければ転送動作の開始位置として TOP_LEFT、$BOTTOM\_LEFT$、そして $RANDOM$、あるいは $SPECIAL\_*$ により、90 度回転、鏡像転置、2 のべき乗倍の拡大または縮小等の転送方法を指定する。$RANDOM$ が指定された場合に限り、別途、開始位置の情報を格納した領域へのポインタを指定する。

(4) **余白属性**：転送対象のデータブロック（後述）以外に、さらに本属性値で指定された量の上下左右の隣接データをも、転送対象に含めたい場合に指定する。例えば 2 次元画像に対する 3x3 のコンボリューション演算では、上下左右それぞれ 1 画素分 (1 バ

イト) の糊しろデータが必要であり、その場合は余白属性値として 1 を指定する。

(5) **放送属性**：work-item 間における参照データの共通性の有無を指定する。共通性がある場合は $COMMON$、共通性がない場合は $SEPARATE$ を指定する。$COMMON$ 属性に指定されたデータ群は、Global 空間と Local 空間との間での転送となる。

(6) **分割属性**：横が WG サイズで縦が分割サイズのブロック (分割データと呼ぶ) への分割に際し、データを各 work-item に割り当てる際に縦優先と横優先のいずれで割り当てるかを指定する (図 5(a) と (b) 参照)。図 5(a) では縦への一方向の点線矢印の長さが一つの work-item への割当データ量 (=分割サイズ) を表し、この分割サイズ分の割当データが、ユーザ処理をコールする前に、AC カーネルによって矢印の方向の順番通りに一つの work-item の Private/Local 空間と Global 空間との間で自動転送される。WG サイズと分割サイズは、後述するように AC カーネルが決定するが、ユーザはその最大値と最小値を指定できる。

(7) **階層属性**：対象データ群を、どのループ階層におくかを、1 から始まる自然数で指定する。

(8) **依存属性**：8 近傍に位置する、他の分割データが自分と参照依存 (LNO/0/1/2)、定義依存 (RNO0/1/2)、または依存無し ($NONE$) のいずれかの関係にあるかを指定する。分割属性が縦優先かつ TOP_LEFT 参照属性が指定されている場合を例にとると、分割データからみて、LNO0 は上下、LNO1 は左、そして LNO1 は左と右の分割データに対し参照依存性を持つ場合、RNO0 は上方、RNO1 は上・斜め上・左、そして RNO2 は 8 隣接のうちの図 4 に示す位置に存在する分割データに対し定義依存性を持つ場合に指定する (図 4 参照)。

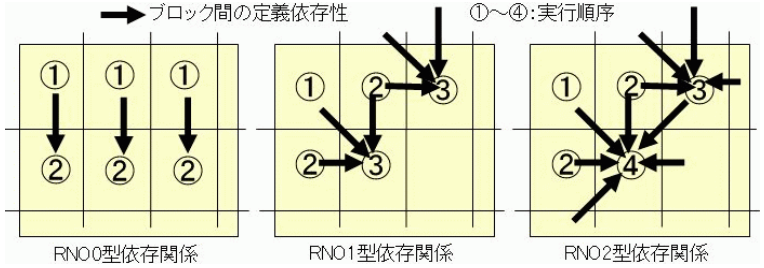

図 4 RNO0〜RNO2 の依存属性の場合の分割データ間の実行順序制約について

3.3 転送シナリオの決定

AC カーネルでは個々の転送対象データ群に対し、ユーザが指定した転送パタンの情報を元に、以下の規則に従って最終的にな転送シナリオを決定する。

(1) 全データ群に対し、共通の WG サイズを設定する。その際、Private/Local 空間の総容量、データサイズ、並びにターゲットマシンの WG 上限制約等を考慮の上で自動決定する。

(2) 階層属性が同一のデータ群が、共通の分割数となるよう、データ群毎に対する分割サイズを決定する。

(3) 階層属性が同一のデータ群は、対応する分割データ毎を、同時に Private/Local 空間に転送してからユーザ処理を起動する。

(4) 階層属性が異なるデータ群同士は、相互の分割データ数の掛け算となる回数 (階層反復数) だけ、対応させた分割データを同時に Private/Local 空間に転送してはユーザ処理を起動する。例えば階層属性が 1 のデータ群の分割数をN、階層属性が 2 のデータ群の分割数をMとすると、AC カーネルはユーザ処理をN×M回だけ呼び出すように動作する。また各呼出しに先立ち、分割データの一種類の組合わせを、Private/Local 空間に転送する。図 5(c) にその動作例を示す。

(5) 分割データの用意以外に、依存属性によって指定された、各分割データと「依存」関係にある他の分割データも、併せて自動転送するように動作する。但し、依存関係が「参照依存」の場合は処理前の分割データ、依存関係が「定義依存」の場合は処理後の分割データが自動転送される。

3.4 ACL フレームワークを利用するための API 群

ACL フレームワークを利用する場合、AC カーネル (AC_kernel) が、OpenCL デバイス側での実行の起点となり、ユーザ処理は AC_kernel からコールされる関数 (User_main) として定義する。

ACL フレームワークのための API 群を以下に示す。ここで、clSetAclKernelArg() と clEnqueueAclKernel() がホスト側の API であり、他はカーネル側 API である。カーネル側 API は、Private/Local 空間へ自動転送されたデータ[⋆1]への参照や、分割サイズ、ローカル分割数、そして現ローカル分割位置など、転送シナリオに関する情報にアクセスするためのビルトイン関数群である。

⋆1 分割サイズ分のデータ、余白分のデータ、依存先参照データ、依存先定義データ等

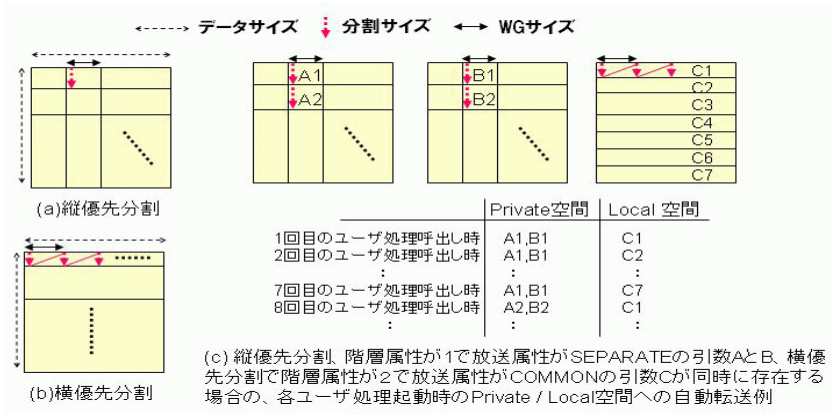

図 5 分割属性と転送シナリオ例 (2 次元データの場合を例に)

- **clSetAclKernelArg()**：ACL フレームワークを利用する場合、clSetKernelArg 関数の代わりに本 API でカーネル引数に対し、転送属性を設定する。
- **clEnqueueAclKernel()**：ACL フレームワークを利用する場合、clEnqueueNDRangeKernel 関数の代わりに本 API でカーネル起動コマンドを Enqueue する。
- **get_acl_blockSze()**： 指定階層の分割サイズを戻す。
- **get_acl_blockNum()**： 指定階層の分割数を戻す。
- **get_acl_levelPos()**： 現階層を戻す。
- **get_acl_levelSze()**： 現階層の反復数を戻す。
- **get_acl_levelIdx()**： 現階層の反復位置を戻す。
- **get_acl_nbh()**：参照依存先データ、定義依存先データ等を格納した Local/Private アドレスを戻す。依存属性が *NONE* 以外の場合に有効。
- **get_acl_lr()**：指定距離 (右が正) に位置する PE への割当データを参照する。本関数を利用する場合は、指定距離以上の余白属性値が指定されている必要がある。

3.5 AC カーネルの処理フロー

AC カーネルは、ユーザ処理の実行の裏に、非同期で次のデータブロックをプリロードすることで、Global 空間と Private/Local 空間の間のデータ移動の遅延を隠蔽するように動作することを基本とする。但し、非同期転送がサポートされていない OpenCL デバイスの場合は、同期転送を行うものとする。図 6 に AC カーネルの処理フローの例を示す。

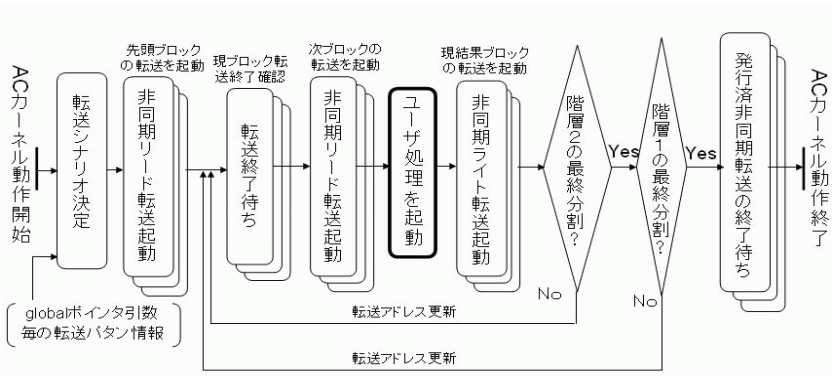

図 6 AC カーネルの処理フロー

3.6 記 述 例

ACL フレームワークの下、画像を 2 値化する処理 (Binarize) について、処理対象と処理結果画像への Global 空間ポインタである src と dst とを、自動転送対象に指定した場合の記述例と転送パタン設定例をそれぞれ図 7 と表 1 に示す。

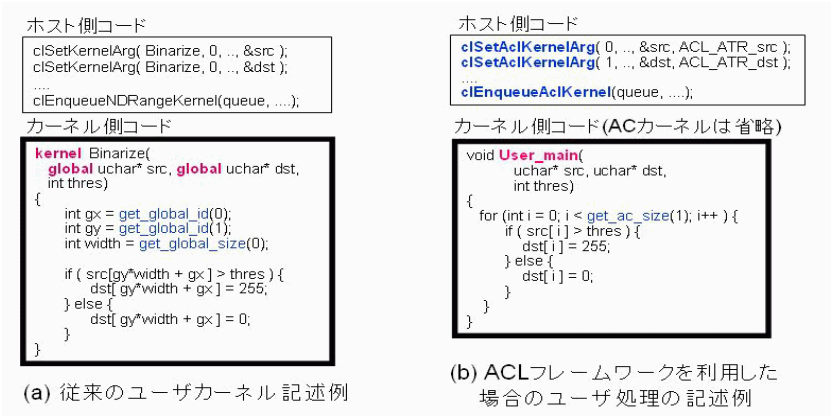

図 7 ACL フレームワーク向け記述例

図 7(a) は Binarize 処理に対する従来の OpenCL カーネル記述例である。ユーザカーネ

表 1 2 値化処理の場合の転送属性設定

ポインター引数名	転送パタン属性							
	サイズ	分割	階層	参照	定義	放送	余白	依存
src	640x480	縦優先	1	TOP_LEFT	NONE	SEPARATE	0	NONE
dst	640x480	縦優先	1	NONE	TOP_LEFT	SEPARATE	0	NONE

ル引数 src,dst は global キーワードにより Global 空間へのポインタとして宣言される。また、src や dst を先頭とする領域へのアクセスアドレスを得るために get_global_id() 等のビルドイン関数を利用している。それに対し ACL フレームワークの場合、ユーザ処理を通常の OpenCL 関数 (User_main) として定義し、そして引数 src と dst は Private 空間へのポインタを受け取るように記述する。src や dst を先頭とする領域をアクセスするには、get_global_id() を元にアドレスを計算する必要はなく、単に連続アドレスを用いればよい。但し従来型の OpenCL カーネルとは、get_ac_size ビルトイン関数を用いて、割当データのサイズ情報を得た上で、個々の割当データに対する処理を反復するように記述する必要がある点で異なる。

4. 予備評価

組込み用途の高並列 SIMD プロセッサ IMAPCAR2-200[1)]、デスクトップ・グラフィックス用途向けに開発された Quadro FX3700[5)]、そして同じ GPU でも GPGPU としての利用を意識した GeForce GTX 580[4)]、の 3 種類の相異なるメモリアーキテクチャを持つアクセラレータ (表 2) を対象に、AC カーネルの一部の機能のみを持つプロトタイプを作成し、２つの簡単な処理タスクを用いて ACL フレームワークの有効性を予備評価した。

表 2 アクセラレータの諸元 (参考)

アクセラレータ名	PE 数	PE 周波数	電力	グローバルアクセス遅延隠蔽法
IMAPCAR2-200[1)]	64PE	133MHz	<1W	ダブルバッファ
Quadro FX3700[5)]	14*8PE	500MHz	<78W	コンテキスト切替
GeForce GTX 580[4)]	16*32PE	1544MHz	<244W	コンテキスト切替+キャッシュ

評価用のタスクとして、図 7 の 2 値化タスクに加え、科学技術計算関連例題として文献[13)]と同様の MDH(multiple Debye-Huckel) 演算タスクを利用した。MDH 演算タスクでは、参照粒子データへのポインタ (ax〜az, charge,size) が指すデータ群に対し、全 work-item が同一位置を同時に参照しながら畳み込み演算を行う。そのため、参照粒子データ群に対しては、放送属性に *COMMON* を指定するのが妥当である。その上、参照粒子データ数が多く、一度では Local 空間に入りきらないため、反復属性指定を利用し、Local 空間内のデータを入れ替えながら畳み込み演算を行うようにした (表 3)。

表 3 MDH 演算タスクの場合の転送属性設定

ポインター引数名	転送パタン属性							
	サイズ	分割	階層	参照	定義	放送	余白	依存
ax,ay,az	256	横優先	2	TOP_LEFT	NONE	COMMON	0	NONE
charge,size	256	横優先	2	TOP_LEFT	NONE	COMMON	0	NONE
gx,gy,gz	1024x1024	縦優先	1	TOP_LEFT	TOP_LEFT	SEPARATE	0	NONE

4.1 各アクセラレータの概要

IMAPCAR2-200[1)] はルネサス製の組込み用高並列 SIMD[⋆1] プロセッサである。各 PE に 4KB の Private メモリ、WG(=64PE 固定) 毎に 4KB の Local メモリが存在する。Local 空間と Global 空間の間はキャッシュ制御によりユーザからは一つのメモリ空間として見えるが、Local 空間アクセスは、全 PE が同一アドレスへアクセスする場合のみ可能である。Private 空間と Global 空間の間のデータ転送には非同期 (DMA) 転送機構が用意されており、Global 空間のアクセス遅延は、Private 空間をダブルバッファ構成とした上で非同期 DMA 転送を行うことにより隠蔽することを基本とする。

FX3700 は NVIDIA 社製 GPU であり、各 PE の Private メモリ容量は不明、Local メモリ容量は 16KB である。Private/Local 空間と Global 空間との間のキャッシュ制御機構はないが、Warp 単位でコンテキストを切り替えるスレッディング機能を備えているため、Global 空間へのアクセス遅延を、ある程度自動的に隠蔽できる。対して GTX580 は Fermi アーキテクチャ[21)] に基づく NVIDIA 社製の最新 GPU である。各 PE の Private メモリは 1KB のレジスタファイル、Local メモリ容量は 64KB、さらにオンチップで 768KB の Global キャッシュを持つ。FX3700 同様に Warp 単位でのコンテキストスイッチが可能な上、キャッシュ制御機構も追加されているため、FX3700 よりも優れた Global 空間アクセス遅延の隠蔽能力が期待できる。但し消費電力もその分だけ高く、FX3700 の 3 倍以上である。

⋆1 MIMD モードも備わるが今回の評価では用いていない

4.2 評価結果

評価では、オリジナルコード[⋆1]、手作業最適化コード[⋆2]、そして ACL(フレームワーク) 利用コード[⋆3]の 3 種類のコードについて、コード行数 (記述煩雑度比較用) と処理時間を測定した。コード行数では、オリジナルコードのそれを 1 とした場合の他のコード比率、処理時間では手作業最適化コードのそれを 1 とした場合の他のコードの比率をそれぞれ求めた結果を図 8 に示す。

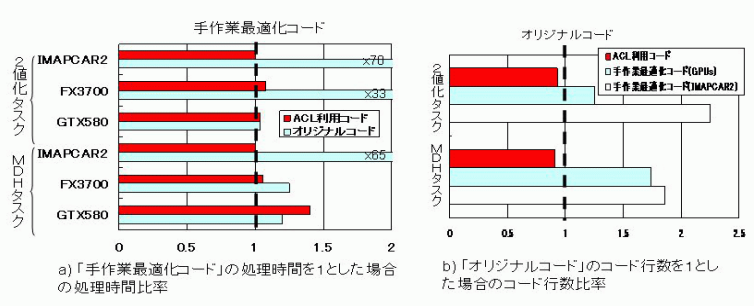

図 8 ACL フレームワークの有効性評価

図 8(a) が処理時間の評価結果である。IMAPCAR2 に対する評価結果では、ACL 利用コードの処理時間がほぼ手作業最適化コードのそれと一致しており、ACL フレームワークの利用に伴うオーバーヘッドはほぼ無視できることがわかる。FX3700 と GTX580 に対する評価結果でも、ACL 利用コードと手作業最適化コードの処理性能がほぼ一致した。一方、FX3700 と GTX580 とで比較すると、FX3700 の場合、ACL 利用コードの方がオリジナルコードよりも、常によりよい性能が得られているのに対し、GTX580 の場合では両者の差が縮まり、MDH 演算タスクでは ACL 利用コードの方が処理時間が若干劣った。これは GTX580 では FX3700 と比べ Global キャッシュが装備されたことで、参照粒子データ群を手作業最適化コードや ACL 利用コードのように、Local 空間へわざわざ移動させなくても、キャッシュヒットすれば、同様の性能が得られることによると思われる。

⋆1 メモリアクセス遅延を考慮せず、直接 Global 空間をアクセスして処理を記述したもの
⋆2 Global 空間へのアクセス回数をできるだけ減らすべく、適宜に Local/Private 空間へデータをコピーしてから演算処理を行うようにオリジナルコードを修正したもの
⋆3 オリジナルコードを元に、本フレームワークの利用を前提に書き換えたもの

図 8(b) がコード行数に関する評価結果である。2 値化と MDH 演算の両タスクで共に、より高性能な ACL 利用コードがオリジナルコードと同等程度のコード行数で済んでいることがわかる。IMAPCAR2 向け手作業最適化コードでは、非同期 DMA 転送に係わる記述が必要となるため、コード行数が大きく増加した。GPU 向け手作業最適化コードでは、特に MDH 演算タスクの場合、Local 空間へのデータ移動を分割して行う指示をユーザコード内で行う必要が発生するため、コード行数が大きく増加した。

5. おわりに

本研究は OpenCL の性能可搬性を改善する手法として、OpenCL カーネル内から Global 空間アクセス記述を排除可能にする ACL フレームワークを提案し、またその利用に向けた基本 API を提案した。

OpenCL が標準的な並列プログラミング言語の座を射止めるためには、少なくとも C 言語と同程度の性能可搬性の実現が重要である。それには、OpenCL デバイス間のアーキテクチャ相違、とりわけ性能への影響が大きいメモリアーキテクチャの相違をうまく隠蔽できる仕組みが必要である。本稿はそのための一つの仕組みを示した。Global 空間アクセス遅延の隠蔽をダブルバッファ方式で行う IMAPCAR2-200 と、コンテキストスイッチ方式で行う 2 種類の GPU とを用いた予備評価では、いずれのアクセラレータに対しても、AC カーネルと Global 空間アクセス記述を排除した ACL 利用コードの組み合わせで、手作業で最適化したコードと同程度の処理性能が得られ、かつ手作業で最適化したコードと比べ記述行数を大幅に削減できることを確認した。今後は、より多様な OpenCL カーネルを対象に、ACL フレームワークの有効性評価を継続して進めていく予定である。

参考文献

1) S. Kyo, et.al:*IMAPCAR2: A Dynamic SIMD/MIMD Mode Switching Processor for Embedded Systems*, Proc. of Hot Chips 21 (2009)
2) *http://www.clearspeed.com/products/csx700.php*
3) *http://www.cognivue.com/docs/PB-10157-00%20CV220X%20Product%20Brief.pdf*
4) *http://www.nvidia.co.jp/object/product-geforce-gtx-580-jp.html*
5) *http://support.express.nec.co.jp/teci/tecbook-pdf/50Tb200912/FX3700rev2.pdf*
6) *The OpenCL Specification*, Khronos Group, Ver.1.0 (2009).
7) S. Antao, L.Sousa: *Exploiting SIMD extensions for linear image processing with OpenCL*, IEEE Int. Conf. on Computer Design (ICCD), pp.425-430 (2010).
8) D. Razmyslovich, et.al.: *Implementation of Smith-Waterman Algorithm in*

OpenCL for GPUs, 2nd Int. Workshop on Parallel and Distributed Methods in Verification, pp.48-56 (2010).
9) 設楽他:*OpenCL* **互換アクセラレータのマルチノード環境における開発負担軽減のためのミドルウェアの実装**, 情報処理学会研究報告, Vol. 2010-HPC-128, No.22, pp.1-8, 2010.
10) 島田他:*OpenCL* **を用いた異種** *GPU* **における性能特性に応じた最適化**, 情報処理学会研究報告, Vol. 2010-HPC-128, No.23, pp.1-7, 2010.
11) *http://www.opencldev.com/*
12) Cho, S. et.al.: *OpenCL and parallel primitives for digital TV applications*, IBM Journal of Research and Development, Vol.54, No.5, pp.1–14 (2010).
13) J.E.Stone, et.al.: *OpenCL: A Parallel Programming Standard for Heterogeneous Computing Systems*, Computing in Science & Engineering, Vol.12, No.3, pp.66-73 (2010).
14) K. Asanovic, et. al: *The Landscape of Parallel Computing Research: A View from Berkeley*, Technical Report No. UCB/EECS-2006-183 (2006).
15) F. Jacob, et.al: *CUDACL: A tool for CUDA and OpenCL programmers*, 2010 International Conference on High Performance Computing (HiPC), pp.1-11 (2010)
16) Tianji Wu, et.al: *Gemma in April: A matrix-like parallel programming architecture on OpenCL*, Conference & Exhibition (DATE) of Design, Automation & Test in Europe, pp.1-6 (2011)
17) Jens Breitbart, Claudia Fohry: *OpenCL - An effective programming model for data parallel computations at the Cell Broadband Engine*, Workshops and Phd Forum of 2010 IEEE International Symposium on Parallel and Distributed Processing (IPDPSW), (2010).
18) B.Sharma, N.Vydyanathan: *Parallel discrete wavelet transform using the Open Computing Language: a performance and portability study*, Workshops and Phd Forum of 2010 IEEE International Symposium on Parallel and Distributed Processing (IPDPSW), (2010).
19) R. Weber, et.al: *Comparing Hardware Accelerators in Scientific Applications: A Case Study* IEEE Transactions on Parallel and Distributed Systems, Vol.22, Issue.1, pp.58-68 (2011).
20) E. Christophe, et.al:*Remote Sensing Processing: From Multicore to GPU*, IEEE Journal of Selected Topics in Applied Earth Observations and Remote Sensing (2010).
21) *http://www.nvidia.com/content/PDF/fermi_white_papers/D.Patterson_Top10InnovationsInNVIDIAFermi.pdf*