Vol.2015-ARC-215 No.3
Vol.2015-OS-133 No.3
2015/5/26

FPGAアクセラレーション向けの実行制御の高速化とシステムソフトウェアの検討

坂本 龍一[1,a)]　小柴 篤史[2]　佐藤 未来子[2]　並木 美太郎[2]　近藤 正章[1]　中村 宏[1]

概要：
近年 FPGA を用いたアクセラレーションが注目されている．FPGA による高性能化のアプローチでは FPGA 外のオフチップメモリへのメモリアクセスを抑制することが重要である．そのため，FPGA 内部でデータを受け渡すパイプライン並列や，データのローカリティを生かすためのネストが用いられる．さらに，近年これらの背景を受け，OpenCL 2.0 にてパイプライン並列とネストがサポートされた．一方で，パイプライン並列やネストは細粒度な実効制御が必要であり，従来の実行方式では制御に要するオーバヘッドが発生し，パイプライン並列やネストが有効に利用できない．そこで，本研究では FPGA アクセラレーション環境を対象とし，実行制御を有効に行うためのハードウェアとシステムソフトウェアについての検討を示す．

1. はじめに

近年，プロセッサの演算性能の向上のためにアクセラレータの利用が注目されている．これらの例として GPU や MIC, FPGA などの利用があげられる．一方で GPU や MIC, 多くの FPGA によるアクセラレーションにおいては，オフチップメモリへのメモリアクセスがボトルネックとなることが大きな課題となっている．粗粒度なデータに対してデータ並列を行うことにより演算効率を向上させているため，オンチップメモリ内のデータでは容量が足りず，オフチップメモリへのアクセスが集中する．一般にオフチップへのメモリアクセスは，オンチップのメモリアクセスに比べ帯域が狭いため，性能が大きく低下する．そのため，多くのデータ並列を行うアプリケーションにおいて性能が悪化する．

そのため，オフチップメモリへのアクセスを削減することが大きな課題となっている．大きく分けパイプライン並列とネストによる 2 つのアプローチがとられている．1 つ目はオフチップへのメモリアクセスを抑制する，タスク並列やパイプライン並列を用いるアプローチである．パイプライン並列ではアクセラレータの各コアに対し，異なるタスクを割り当てコア間でデータを受け流すことによりオフチップへのデータアクセスを抑制する [1]．また，これらのアプローチは FPGA において有効に利用される．FPGA にてデータパスをハードウェア化することにより，高い演算性能を実現する．2 つ目はデータのローカリティを生かすためにネストを用いるアプローチである．ネストを用いて入力データに対し細かな演算を行うことにより，時間的な局所性を高め，キャッシュを有効に利用しオフチップへのメモリアクセスを抑制する．

また，近年これらのプログラミングを行うための実行環境が大きく改善されている．汎用の並列プログラミング言語の OpenCL[2] では旧来データ並列とタスク並列のみをサポートしていたが，OpenCL 2.0 からは新たにパイプライン並列とネストのサポートが行われている．

パイプライン並列では FIFO 機能を持つメモリオブジェクトの Pipes が拡張された．ホストコードで本 Pipes を利用することで，簡潔にパイプライン並列処理が記述できる．また，ネストにおいてはカーネル中から他のカーネルのキューイングができるように改良された．従来カーネルの実行はホストコード側からのみ制御可能であり，ネストを行う際は，カーネルの演算結果をいったんホスト側にコピーし，ホストが改めて最適なカーネルをキューイングする必要があり，オーバヘッドが大きかった．OpenCL 2.0 からはカーネルから他のカーネルのキューイングを行うことが可能となった．これにより，パイプライン並列やネストを用いることで，オフチップメモリへのメモリアクセスを抑制できる．

[1] 東京大学
東京都文京区本郷 7-3-1
[2] 東京農工大学
東京都小金井市中町 2-24-16
[a)] r-sakamoto@hal.ipc.i.u-tokyo.ac.jp

一方で，これらのパイプライン並列やネストを効率よく実行するためには，ランタイムな実行制御がオーバヘッドとなる課題がある．パイプライン並列ではオンチップの小規模（数 KB〜十数 KB）な SRAM を FIFO として利用するため，アクセラレータで動作するタスクの実行時間が短く（数百クロックから千クロック程度）なり，データ並列と比較して，アクセラレータや FIFO に対する細粒度な実行制御が必要となる．また，ネストにおいてもキャッシュを有効に利用することを目指すため，カーネルで扱うデータ量が小さくなり，パイプライン並列と同様に細粒度な実行制御が必要となる．これらの実行制御をソフトウェアにて行った場合，数百から 2 千クロック程度のオーバヘッドが生じる．その結果，実行制御に要するオーバヘッドが増加し，実効性能が悪化する課題がある．

そこで，本研究では実効制御の高速化を目指す．具体的には，パイプライン制御やネスト制御などのランタイムな実効制御を高速化するための，ハードウェア タスク ディスパッチャ (RCH) を提案する．アクセラレータや DMAC，FIFO へ対する実行制御をハードウェアにて高速化するための構成を明らかとする．実行制御を高速化することで，演算性能を改善する．さらに，近年注目されている FPGA によるアクセラレーション環境への適用方法を明らかとする．

2 章では OpenCL によるパイプライン並列やネストの特徴・課題を明らかとし，さらに，これらを受け本研究の目標とアプローチを示す．3 章では提案するシステムの全体構成を示す．4 章ではパイプライン並列制御を隠蔽する OpenCL ライブラリの設計を示す．5 章では実行制御の高速化を行う RCH の設計を示す．6 章では実装について述べ，7 章で評価を示す．8 章で関連研究について示す．9 章にてまとめを述べる．

2. 本研究の課題と目標

OpenCL 2.0 で拡張されたパイプライン並列とネストを用いることで，FPGA アクセラレーションのメモリボトルネックを解消できるが，実行環境が課題となる．そこで，課題を明らかとし，本研究の目標を示す．

2.1 実行制御オーバヘッド

本節では FPGA アクセラレーションを対象に，OpenCL を用いてパイプライン並列とネストを行う際の課題を示す．

(1) FPGA 環境におけるネストの課題

入力データがランタイムに変化し，入力データに依存し演算のタスクグラフが変わる場合などに，ネストが有効に利用できる．OpenCL 2.0 では，カーネル内から，他のカーネルをネストすることができるようになっている．旧来の OpenCL ではネストを行う際には，いったんカーネルで演算したデータをホストにコピーし，改めて，ホストが最適なカーネルを決定し，コマンドキューに対しタスクのエンキューを行う必要があり，オーバヘッドが大きかった．これが，OpenCL 2.0 では，カーネル内から他のカーネルをネストすることができるようになっており，データのコピーがなくなりオーバヘッドが改善されている．

一方で，FPGA アクセラレーションにおいては 2 通りのネストの方法が考えられる．1 つ目がネストされるすべてのタスクグラフを展開し，それらを，すべて回路に展開する方法である．この方法は単純であるが，タスクのノードの利用頻度が少ない部分があり，回路リソースに無駄が多くなる問題がある．2 つ目は複数のカーネルで共通に用いる演算をサポートするアクセラレーションモジュールを複数持ち，適宜，パラメータを切り替える方法である．これにより，回路リソースを抑えることが可能となる．ここでは，回路リソースを優先し，共通の機能を持つアクセラレーションモジュールを適宜ネスト先として利用することを考える．

この場合，ネストする際に入力データなどに応じ，アクセラレーションモジュールを適宜起動する制御が必要となる．すなわち，ランタイムに適切なアクセラレーションモジュールを利用することが重要となる．

さらに，一般にネストされる演算はキャッシュやオンチップメモリを有効に利用することを目指すため，演算の粒度が細粒度になる．そのため，ソフトウェアにてネスト先のアクセラレーションモジュールの起動や，アクセラレーションモジュール間での同期を行った場合ソフトウェアによる制御がオーバヘッドとなる．ネスト先のカーネルが実行に要するクロックサイクルは数十クロックサイクルから数百クロックサイクルであるのに対し，システムソフトウェアを用いた制御には数百クロックサイクルから 2000 クロックサイクル程度の時間を要し，性能が大きく悪化する．

(2) パイプライン並列の実行制御時の課題

パイプライン並列ではカーネル間の演算データの受け渡しに小規模なオンチップメモリを FIFO(数 KB〜十数 KB) として用いる．そのため，必然的に各アクセラレーションモジュールで実行するタスクの実行時間が短くなる（数百クロックサイクル〜千クロックサイクル程度）．これは，データ並列を得意とするメニーコアプロセッサと比較し，細かい粒度である．そのため，ネスト時と同様にソフトウェアによるパイプライン並列の実行制御が大きな課題となる．

2.2 FPGA 向け OpenCL 実行環境

これにより，パイプライン並列の制御とネスト制御を高速化し実行性能の向上を目指す．そこで，本研究ではこれらの目標を達成するために 2 つのアプローチをとる．

- ハードウェア タスク ディスパッチャによる実行制御

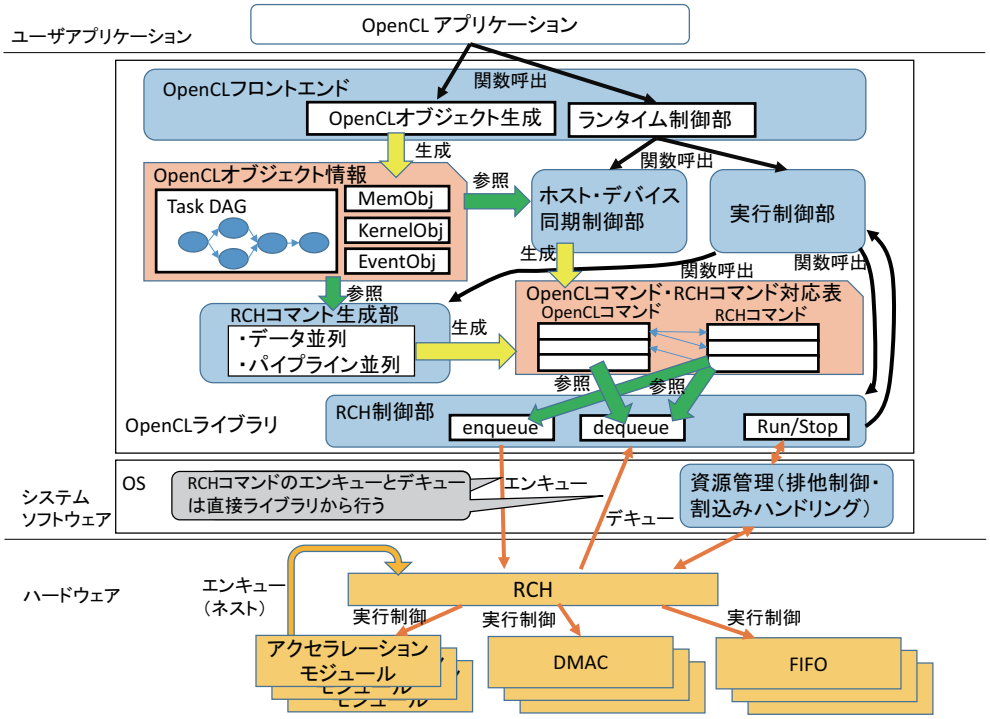

図 1　提案システムの全体構成

の高速化
パイプライン時のアクセラレーションモジュールの起動，FIFO による同期制御，ネストによるアクセラレーションモジュールの起動をハードウェアによって行うことにとり，ソフトウェアの実効制御オーバヘッドを改善し演算性能を向上する．本研究では本ハードウェア タスク ディスパッチャを，FPGA 向け RCH として提案する．

- OpenCL ライブラリによるハードウェア タスク ディスパッチャ制御の隠ぺい
高速化を行うハードウェアを提案したとしても，専用のハードウェアのために，ハードウェアの仕様に応じて OpenCL を拡張するアプローチを行うと，プログラミングインタフェースの汎用性が損なわれる．そのため，本研究ではハードウェアタスクディスパッチャを OpenCL に隠ぺいする．

これらにより，OpenCL による FPGA の実行制御の高速化を目指す．また，本環研究では FPGA アクセラレーション環境においてカーネルの実効制御・同期制御・データ転送制御を効率よく行うことに着目する．そのため，カーネルコードは既存の高位合成言語環境等を利用し，アクセラレーションモジュールに合成されることを想定する．

3. システムの全体構成

FPGA アクセラレータの実行性能の向上を目指し，RCH による実行制御の高速化と，OpenCL ライブラリによる FPGA と RCH の隠ぺいを実現する．提案するシステムの概要を図 1 に示す．

3.1 システムの概要

OpenCL ライブラリはアプリケーションプログラマに対してデータ並列・タスク並列・Pipes や Nested Parallel の機能を提供し，RCH を含む FPGA ハードウェアを隠ぺいする．また，RCH はソフトウェアにて行っていた実行制御をハードウェアにて高速に実現し，アクセラレータの実行効率を改善する役割を担う．

3.2 OpenCL ライブラリの役割

OpenCL ライブラリは主に 2 つの役割を担う．1 つ目はアプリケーションプログラマに対して抽象化された OpenCL デバイスを提供することである．アプリケーションプログラマはこれらの抽象化された OpenCL デバイスを用いてデータ並列制御・タスク並列制御・パイプライン並列制御を記載する．これらの情報を OpenCL オブジェクト情報として管理する．また，ネストはカーネル内の API に抽象化する．2 つ目は，RCH の実行制御を行う役割である．RCH の起動や停止などの基本的な役割を担う．

3.3 OS の役割

本研究では実行制御の高速化を目指している．そのため，極力アクセラレーションモジュール制御や RCH 制御に OS は関与しないようにする．具体的には，RCH への実行依頼を行う際，OpenCL ライブラリは OS をかえさずに直接 RCH への実行を依頼する．このようにして OS に遷

移する際の時間的なオーバヘッドを削減する．一方で，多数のユーザアプリケーションからの RCH 利用に対する排他制御や，RCH からの終了やエラーなどのハンドリングは OS で行う．

3.4 RCH の役割

RCH はソフトウェアによる実行制御を高速化する役割を持っている．具体的には個々のアクセラレーションモジュールの実行制御や個々の DMAC へ対するデータ転送制御，FIFO 制御，アクセラレータ間同期をハードウェアにより高速に行う．また，アクセラレーションモジュールの機能の入れ替えのため，アクセラレーションモジュールのプログラムメモリの修正や動的再構成を行う．これにより，従来ソフトウェアで行っていたタスクグラフに対するシーケンス制御をハードウェアにてプログラマブルに行うことで，制御を高速化する役割をもつ．

3.5 OpenCL ライブラリと RCH 間のインタフェース

OpenCL ライブラリと RCH 間はデータ並列演算・パイプライン並列演算・ネスト制御をプリミティブ化した RCH コマンドにてインタフェースをとる．RCH コマンドはアクセラレーションモジュール制御や FIFO 制御，DMAC に対する制御を抽象化したものである．この RCH コマンドによって，ホストプロセッサと非同期にアクセラレーションモジュール制御や FIFO 制御が可能となり，ホストプロセッサと RCH 間で並列性を向上させることが可能となる．これらの RCH コマンドを Submission Queue へ投入することで，RCH を制御する．また，RCH は RCH の終了時に終了コマンドを Completion Queue に投入し，RCH コマンドの終了を非同期に通知する．

また，RCH の演算の終了時やエラー時には RCH とホストプロセッサ間での同期が必要である．そのため，RCH にはホストプロセッサと同期を取る同期コマンドを持つ．同期コマンドを用いた同期やエラーの際の同期では，RCH がホストプロセッサに対して割込を行うことにより，同期を行う．

さらに，本 RCH はアクセラレーションモジュールの機能の切り替えのためにプログラムメモリの更新とアクセラレーションモジュールの動的再構成を行うためのコマンドを有する．これにより，システムソフトウェアの関与なしに，アクセラレーションモジュールの機能の切り替えを可能とする．

4. OpenCL による抽象化と OpenCL ライブラリの設計

OpenCL ではアクセラレータを OpenCL デバイスモデルに抽象化し，データ並列やパイプライン並列，ネストを並列モデルとして提供しいている．この OpenCL デバイスモデルに対して，並列モデルによる演算を行うことで，煩雑な実効制御が隠ぺいされる．次に，OpenCL による FPGA の抽象化について示す．

また，対象とする FPGA 環境はプロセッサ内に ARM コアと FPGA を搭載するようなヘテロジーニアスな環境を対象としている．さらに，FPGA 内のは複数のアクセラレーションモジュールから構成され，アクセラレーションごとに性能が異なるヘテロなアーキテクチャを想定する．

4.1 デバイスの抽象化

FPGA 向けの OpenCL によるデバイスの対応を表 1 に示す．

表 1 OpenCL デバイスモデルと FPGA 対応

OpenCL デバイス	FPGA 内のブロック
ホストプロセッサ	ARM プロセッサ
デバイス	FPGA 部
コンピュートユニット	アクセラレーションモジュール
ホストメモリ	オフチップの大容量メモリ
デバイスメモリ	オフチップの大容量メモリ
ローカルメモリ	ブロック RAM, オンチップ SRAM

4.2 データ並列演算による抽象化

OpenCL ではデータ並列を clEnqueueNDRangeKernel ランタイム API から自動的に行うことができる．データ並列を行う際は，OpenCL ライブラリは利用可能なアクセラレーションモジュールに対し適宜データ分割を行い，各アクセラレーションモジュールに対するデータコピー，演算実行を WRITE, READ, EXE RCH コマンドとして発行する．

4.3 パイプライン並列演算による抽象化

OpenCL でパイプライン並列を行う場合は，FIFO 機能を持つ Pipe メモバッファに対して，カーネル中から write_pipe, read_pipe することにより FIFO に対する put と get 操作を行うことができる．これらの put 操作・get 操作を元に，PUT, GET RCH コマンドを生成する．

4.4 ネストによる抽象化

ランタイムに入力データが変化し，かつ入力データによってタスクグラフが変わる場合，カーネルないからデータに応じて新たなカーネルの実行をネストすることができる．カーネル内で enqueue_kernel API を呼び出すことで，アクセラエーションハードウェアは新たなカーネルの実行を RCH に依頼する．この際は，ネスト向けの RCH コマンドを生成する．

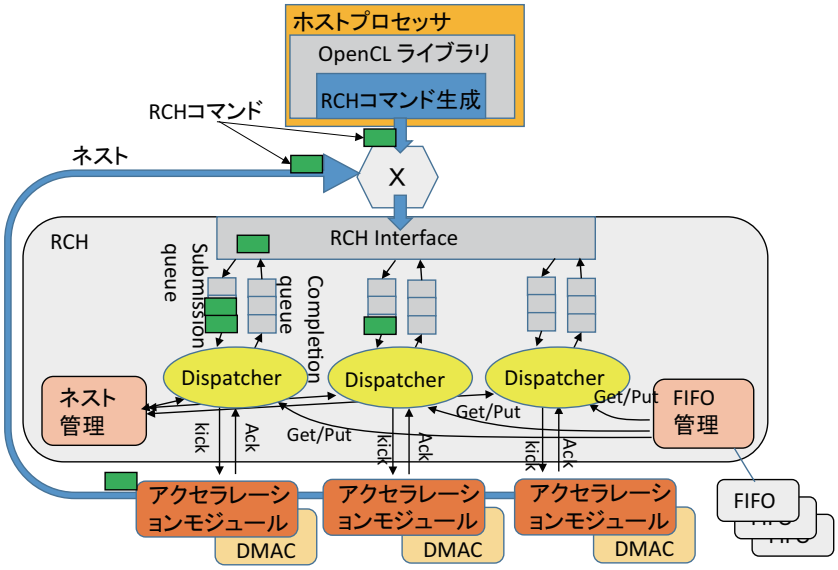

図 2　RCH の概要
Fig. 2　Overview of RCH

5. RCH による実行制御

RCH はソフトウェアによる実行制御を高速化する役割を持っている．具体的には個々のアクセラレーションモジュールの実行制御や個々の DMAC へ対するデータ転送制御をハードウェアから高速に行う．RCH の概要を図 2 に示す．

5.1 パイプライン並列への対応

パイプライン並列を行うためにはそれぞれのアクセラレーションモジュールが異なるタスクを実行し，それぞれ，独立して動作する必要がある．そのために RCH では 1 つ 1 つのアクセラレーションモジュールを独立して動作させるために，アクセラレーションモジュールに 1 対 1 に対応するディスパッチャから構成される．それぞれの，ディスパッチャは対応するアクセラレーションモジュールの実行制御を行う．さらに，DMAC も合わせて制御する．

また，パイプライン並列を行う際は，2 つのアクセラレータ間で FIFO を用いてデータのやり取りを行っている．FIFO を用いることで 2 つのアクセラレータ間での非同期なタスク実行をサポートする．一方で，FIFO full や FIFO empty により 2 つのアクセラレータ間の同期を保証する役割も持つ．

そこで，本研究ではこれらの FIFO の管理をハードウェアとして一括して行う．FIFO の状態をハードウェアにて一括管理することでアクセラレーションモジュール間での FIFO を用いた同期制御を簡単化する．

さらに，これらの FIFO に対する put/get 制御も RCH から行うことにより高速化を実現する．具体的には，FIFO を制御する GET/PUT RCH コマンドを用いる．ディスパッチャは GET コマンドを受け取った場合，FIFO 管理モジュールに対して GET 制御を要求する．FIFO 管理部では FIFO のデータの有無を確認し，有効なデータが無ければ，ディスパッチャをロックする．有効なデータがある場合は，ロックが解除される．同様に，ディスパッチャが PUT コマンドを受け取った場合は，FIFO 管理部に PUT 制御を要求する．FIFO に空きが無ければディスパッチャはロックされ，空きがあればロックが解除される．このように，ハードウェアにてアクセラレーションモジュール制御，データ転送制御，FIFO 制御を行うことで制御オーバヘッドを大きく改善する．

5.2 ネストへの対応

パイプライン並列処理の際は，ホストコードから RCH を制御する．すなわち，ホストプロセッサが RCH コマンドを生成し，RCH がアクセラレーションモジュールを制御する．一方で，ネスト時はアクセラレーションモジュールが RCH に対して，RCH コマンドを生成することにネストを実現する．ネストの制御にはネスト元のアクセラレーションモジュールから RCH に対して NEST_EXE コマンドを生成する．その後，さらに，NEST_WAIT によって同期を図る．

5.3 RCH のコマンドセット

RCH ではパイプライン並列制御に用いる基本的な実行制御を抽象化した RCH コマンドを持つ．これらをのコマンドを表 2 に示す．

NEST_EXE, NEST_WAIT 以外の RCH コマンドは OpenCL ライブラリが，アプリケーションプログラマが記載したデータ並列処理，タスク並列処理，パイプラ

イン並列処理を元にし，RCH に対して発行する．また，NEST_EXE, NEST_WAIT はアクセラレーションモジュールが発行する．これにより，ソフトウェアボトルネックとなる実行制御を，RCH にオフロードすることで実行制御オーバヘッドを改善し，演算効率を改善する．

6. 実装

図 3 にハードウェア実装の概要を示す．

本研究では実装に Xilinx 社の Zynq を用いた．初めに簡単に Zynq の概要を示し，その後，RCH を含むハードウェアの実装，システムソフトウェアの実装について示す．

6.1 Zynq

Zynq は FPGA と ARM 社の Cortex-A9 を搭載した SoC である．Zynq はソフトウェア資源とハードウェアアクセラレーションを同時に利用できる特徴を持っている．そのため，本実装では OpenCL ライブラリを ARM 上で動作するソフトウェアとし，FPGA 部に RCH やアクセラレーションモジュール，各種バス等を実装した．

6.2 ハードウェアの実装

RCH やアクセラレーションモジュール部は Zynq の FPGA 部に実装を行っている．まず，Zynq の FPGA 部にあたる Programmable Logic 部に RCH, DMAC, アクセラレーションモジュールを実装している．ワーキングスペースとしてローカルメモリや FIFO をブロックラムを用いて実装している．RCH はソフトウェアに代わり DMAC やアクセラレーションモジュールの実効制御を行う．また，アクセラレーションモジュールや DMAC は RCH からの制御を受け，演算・データ転送を行う．演算を行う際は，DMAC を用いてあらかじめ FIFO やローカルメモリへデータをコピーしたのち，演算に利用することでメモリアクセスレイテンシを隠ぺいする．さらに，マルチチャネルのメモリにすることでメモリへの帯域を広げるようにした．また，Zynq は ARM の L2 キャッシュと Programmable Logic 間でキャッシュのコヒーレンシを保つための機能を有しているため，これらの機能を利用し，キャッシュを有効に利用できるようにしている．これにより，演算を行うデータや RCH コマンドへのメモリアクセスが改善されることが期待できる．

表 2　RCH コマンドの一覧

コマンド名	コマンドの機能
EXE	アクセラレーションモジュールを起動する
READ	DMAC を用いてオンチップのデータ オフチップへコピーする
WRITE	DMAC を用いてオフチップのデータを をオンチップへコピーする
GET	FIFO からデータを取り出す
PUT	FIFO へデータを詰める
NEST_EXE	アクセラレーションモジュールを起動する．
NEST_WAIT	ネストし先と同期する
SYNC_HOST	ホストプロセッサと同期を行う

6.2.1 パイプライン制御

パイプライン時はローカルメモリを FIFO として利用する．FIFO を構成するためのアドレス計算やフラグ制御は RCH 内の FIFO 管理部にて行う．RCH 内部のディスパッチャはこれらの FIFO を利用して，アクセラレーションモジュールが演算を行うように RCH コマンドをベースに実効制御を行う．

6.2.2 ネスト制御

ネストを行う際はアクセラレーションモジュールが RCH のバスマスタデバイスとなり，RCH に対して NEST RCH コマンドを生成する．RCH は生成された NEST RCH コマンドを受けると，Dispatcher が新たにアクセラレーションモジュールの起動を指示する．さらに，ネストされたアクセラレーションモジュールでの演算が終了した場合，ネスト元に通知する．また，ネストされたカーネルとは，ローカルメモリを共有のメモリとして演算を行う．

6.3 システムソフトウェアの実装

本実装では，ARM 上で直接動作する C 言語で記載したプログラムとして OpenCL ライブラリを実装した．また，RCH の実行制御や，RCH コマンドの挿入や終了コマンドの取出しなどは，アドレスマップされたハードウェアを OpenCL ライブラリからを直接制御する環境として試作した．Linux を用いた環境は実装中である．

また，ARM と RCH は非同期に動作することが可能である．そのため，OpenCL ライブラリと RCH を独立に動作させ並列度を改善することができる特徴を持つ．一方で，本システムソフトウェア環境に置いては，RCH の性能向上，OpenCL のオーバヘッドの基礎評価を目指している．そのため，OpenCL フロントエンド，RCH による実行制御，終了コマンドの取出し，OpenCL の後処理をすべてシーケンシャル行い，それぞれに要するオーバヘッドを確認する．OpenCL ライブラリと RCH の並列動作については今後の課題とした．

7. 評価

本評価では RCH による性能向上についての評価を行い，ハードウェアによる実効制御の高速化の有効性を確認する．本評価では実行制御をソフトウェアから行った場合と，RCH から行った場合の実行時間についての比較を示す．また，現在 Zynq 環境への実装を進めているが，デバッグを行っている最中でありベンチマーク全体としての実行時間を計測できていない．個々の機能の動作は確認ができているため，これらの実測値を元に，演算に要する時間を

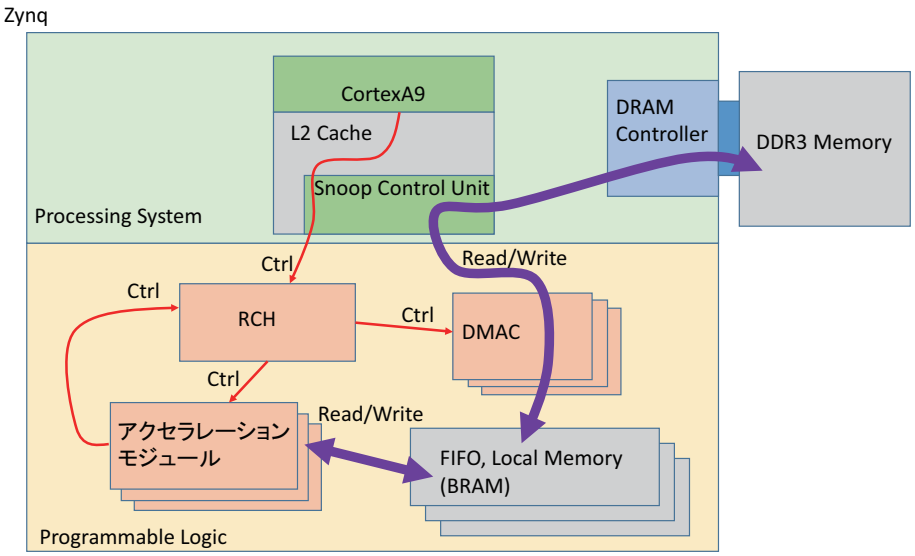

図 3　Zynq 環境における RCH 実装

見積もることとした．実機で全体を通しての評価は今後の課題である．

7.1　評価環境

本評価では Xilinx 社の Zynq 評価ボードの ZC702[7] を利用した．また，そのほかのパラメータを表 3 に示す．

表 3　評価に用いたパラメータ

ARM の動作周波数	667Mhz
RCH の動作周波数	200Mhz
アクセラレーションモジュールの動作周波数	200Mhz
DMAC の動作周波数	200Mhz

7.2　RCH による制御の高速化

次に，RCH による実効制御の高速化についての評価を示す．評価内容としてパイプライン並列環境における比較と，ネストを用いた場合の比較を示す．

7.2.1　パイプラインの評価

パイプライン並列の評価では，パイプラインを利用した 2 つのアプリケーションにおける総演算時間をの比較を示す．ここでは，色変換の Blender と JPEG デコードについて，ARM 上のソフトウェアで制御した場合（soft）と，RCH にて制御を行った場合の比較を示す．本評価ではアクセラレーションモジュールによる演算時間と制御時間の和を総演算時間としている．また，制御オーバヘッドの削減効果を明らかとするため，制御時間を 0 とした理想状態 (ideal) との比較を示す．本比較では理想状態を 1 として正規化を行っている．これらの比較結果を図 4 に示す．Blender ではソフトウェアによる実効制御を RCH にて行うことで，総演算時間を 5%程度高速化することができた．また，jpeg については 14%の高速化を実現した．jpeg が

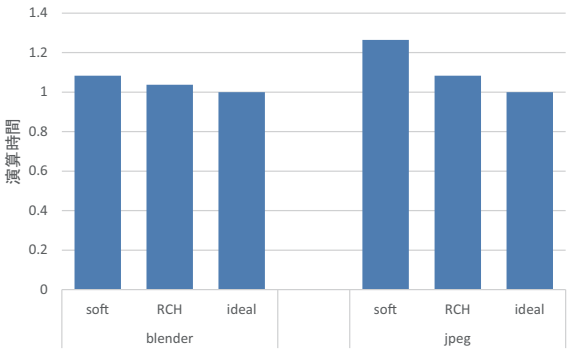

図 4　パイプライン並列時のオーバヘッド

改善率が高いのは Blender と比較し，カーネルの実行時間が短く，相対的に制御に要する時間が大きいためである．一方で，RCH を用いても ideal と比較した場合，4%から 7%程度の制御オーバヘッドが残っている．これは，AXI バスの遅延が大きな原因であると考えられる．

本評価では初期評価のためパイプライン段数の小さな環境において評価を行っている．そのため，さらにパイプライン段数の長い実用的なアプリケーションにおいては，さらに制御オーバヘッドが大きくなると考えられるため，本手法は有効といえる．

7.2.2　ネストの評価

ネストにおける評価では，ネストされたカーネルの時間を振った場合の比較を示す．具体的にはネストされたカーネルの実行に要するクロック数を 100，500，1000 とした場合の比較を行った．図 5 に比較結果を示す．この結果，ネストされるカーネルの実行に要するクロック数が 100，500，1000 の場合のそれぞれで 20%, 7%, 3%の制御オーバヘッドを削減することができた．このように RCH にて有効に高速化を実現できたといえる．また，パイプライン時と同様により細粒度な制御に対して RCH は有効に働くこ

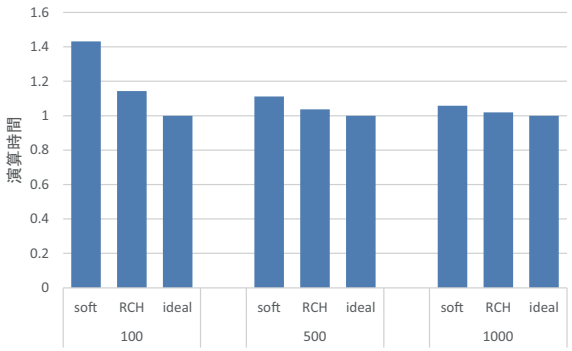

図 5 ネスト時のオーバヘッド（ネストするカーネルのクロック数）

とが分かる．

8. 関連研究

パイプライン並列向けのアクセラレータとして，RAW[3] プロセッサがある．RAW プロセッサでは，算術演算レベルでアクセラレータ間パイプラインを行うことができる特徴を持つ．具体的には，アクセラレータの一部の汎用レジスタが他のアクセラレータの汎用レジスタへ FIFO を介して接続されている特徴を持つ．そのため，ワードレベルでのパイプラインが可能である．しかし，ワード単位ではスループットが悪化する課題がある．また RAW プロセッサは独自のパイプライン言語 StreamIt[6] によるプログラミングをサポートしている．具体的には抽象化された FIFO に対する put/get 制御によりパイプラインを記述する．KALRAY プロセッサ [4] も RAW 同様にパイプライン並列をサポートするアクセラレータである．一方で，パイプライン並列の制御には独自の開発環境が必要であり，煩雑である課題がある．

タスクの実行制御の高速化を目指した研究では，Sanjeev らの研究 [8] が再粒度なタスク（数千クロック以下程度）の高速化を目指し，ハードウェア タスクディスパッチャを提案している．同じく，Daniel らの研究 [9] に置いても，タスクの実行制御を高速化するハードウェアを提案してる．一方で，これらの研究ではタスクスティーリングを用いたタスクスケジューリングに着目しておりパイプライン並列制御は高速に行うことはできない課題がある．

また，演算のアクセラレーションの方法として FPGA が多く普及している．さらに，FPGA 向けの OpenCL 環境も研究，販売 [10][11] され，一般に普及していると言える．一方で，これらはデータ並列に最適化されている．FPGA はストリーミングによるパイプライン制御が非常に優れている特徴を持つが，FPGA 向けの OpenCL 環境ではパイプライン並列は扱えず，FPGA の本来の性能発揮できない課題がある．[10] においてはストリーミング処理のサポートを行っているが独自の API 拡張を行っているため，従来の OpenCL と互換性がない問題がある．

9. まとめ

本研究では FPGA アクセラレーションの制御の高速化をめざし，ハードウェアによる実行効率の高速化手法を提案した．また，OpenCL による FPGA の抽象化方法について示した．ハードウェアによる実行制御の高速化により実行性能を改善することができた．最大で 20%の性能向上を達成することができた．今後は FPGA 向けの OpenCL 環境をより一般化し，ヘテロジーニアスプロセッサへの適用を検討する．

謝辞 本研究は JSPS 科研費 基盤研究 S 25220002 の助成を受けたものである．

参考文献

[1] Hoeseok Yang and Soonhoi Ha, “ILP based data parallel multi-task mapping/scheduling technique for MPSoC”, SoC Design Conference, 2008. ISOCC ’08. International, vol. 01, pp.I-134 - I-137, nov. 2008.
[2] Khronos OpenCL `https://www.khronos.org/opencl/`
[3] Michael Bedford Taylor, Jason Sungtae Kim, Jason E. Miller, David Wentzlaff, Fae Ghodrat, Ben Greenwald, Henry Hoffmann, Paul Johnson, Jae-Wook Lee, Walter Lee, Albert Ma, Arvind Saraf, “The Raw Microprocessor: A Computational Fabric for Software Circuits and General Purpose Programs”, IEEE Micro - MICRO , vol. 22, no. 2, pp. 25-35, 2002
[4] KALRAY: Our MPPA MANYCORE Products `http://www.kalray.eu/products/mppa-manycore/`
[5] Tilera, ”Tilepro64 multicore processor product brief”, `http://www.tilera.com/`
[6] Thies, William and Karczmarek, Michal and Amarasinghe, Saman P. ”StreamIt: A Language for Streaming Applications”, Proceedings of the 11th International Conference on Compiler Construction, 2002
[7] `http://japan.xilinx.com/products/boards-and-kits/ek-z7-zc702-g.html`
[8] Sanjeev Kumar, Christopher J. Hughes , Anthony Nguyen, ”Carbon: architectural support for fine-grained parallelism on chip multiprocessors”, Proceeding ISCA ’07 Proceedings of the 34th annual international symposium on Computer architecture Pages 162-173
[9] Daniel Sanchez, Richard M. Yoo, Christos Kozyrakis, ”Flexible architectural support for fine-grain scheduling”, Proceeding ASPLOS XV Proceedings of the fifteenth edition of ASPLOS on Architectural support for programming languages and operating systems Pages 311-322
[10] Altera, ”アルテラ SDK for OpenCL”, `https://www.altera.co.jp/products/design-software/embedded-software-developers/opencl/overview.html`
[11] Xilinx, ”SDAccel 開発環境”, `http://japan.xilinx.com/products/design-tools/sdx/sdaccel.html`