

Actor モデルにもとづいた非同期並列プログラミング言語 ActGPU のコンパイラの実装とその評価

高柳 亘¹ 徐 駿剣¹ 脇田 建^{1,2}

概要: 我々は以前, NVIDIA の GPU 上で動作するアクター計算の実行時システムを開発し, GPGPU による並列オブジェクト指向プログラミングを実現した. 本研究では, その並列処理系を利用する非同期並列プログラミング言語 ActGPU を定義し, コンパイラを実装した. ActGPU を用いることでプログラマは, 通常の GPGPU プログラミングでは必要とされる並列制御命令の挿入, SIMD 実行方式を意識したスレッドスケジューリングなどの煩瑣な処理を非同期メッセージ送信という単純かつ統一的な枠組みに置き換えることができる. 本論文は, ActGPU 言語の仕様, コンパイラの構成を説明し, その記述力をプログラミング例などを用いて議論する.

キーワード: アクター計算, メッセージ通信, CUDA, コンパイラ

An Implementation and Evaluation of a Compiler for ActGPU, An Actor-Based Asynchronous Parallel Programming Language

TAKAYANAGI WATARU¹ XU JUNJIAN¹ WAKITA KEN^{1,2}

Abstract: In our previous work, we proposed a parallel runtime system based on the Actor parallel computation model on NVIDIA GPU, which allows the programmer to describe parallel programs in object-oriented paradigm. We extend this work to define ActGPU, the asynchronous parallel programming language, using the previous system, and to implement a compiler for this language. ActGPU provides users with message passing as a highly abstract programming construct and a unified tool to describe synchronization and parallelism. By using ActGPU, programmers are freed from complex and error-prone synchronization coding using CUDA's low-level synchronization directives or task scheduling policy of multi-layered thread architecture with the message passing system. This paper explains the specification of ActGP and structure of compiler, and discusses the descriptiveness with ActGPU sample code.

Keywords: Actor model, message passing, CUDA, compiler

1. はじめに

近年 GPU を汎用計算に利用する, *General Purpose computing on GPU (GPGPU)* という技術は, 多くのアプリケーションにおいて優れた計算性能を示している. 元来画像データ処理のために, GPU は内部に大多数の計算ユニッ

トを持っており, それらが超並列に計算を行うことで, 高速な処理を可能としている.

しかし実際は, プログラマは GPGPU プログラミングを行うにあたって多くの努力をしなければならない. 一般的な並列処理においても言えることだが, 正しい結果を出力させるために並列計算ノードごとの通信または同期を考慮に入れる必要があり, より優れた性能を出すためにロードバランスやデータの移動経路に注意を払わなければならない. CUDA[7] や OpenCL[8] は GPGPU アーキテクチャを利用した並列処理言語としてよく知られている. これらの

¹ 東京工業大学大学院情報理工学研究科
Graduate School of Information Science and Engineering,
Tokyo Institute of Technology

² JST CREST
Japan Science and Technology Agency

言語を用いることで簡単に GPGPU プログラミングを行うことができるが、やはりプログラムの正しさ、性能を出すなどの問題を解決するためにはプログラマによる調整が必要となる。そのため、GPGPU は敷居の高い技術であると言える。

本研究のテーマは**アクターモデル**を GPU アーキテクチャに適応させることで、簡単な記述で一定の性能をもたらす GPGPU プログラミングを可能とすることである。アクターモデルは、並列・分散環境でのプログラミングにおいて広く用いられる**メッセージパッシング方式**に基づく理論的、実践的でシンプルなモデルであり、本質的な同時性を備えるため、並列プログラミングで必要となる同期などを考慮せずにプログラミングを行うことを可能とする [2], [3], [4]。またモデル導入により加算される実質的なオーバーヘッドは、メッセージパッシング機構によるもののみであるため、デメリットは小さい。この性質を享受することで、容易で高性能な結果をもたらす GPGPU プログラミングが可能になると我々は考えた。

先の研究として、我々は CUDA を用いたアクター計算の実行時システムを構築した。同期、排他制御などを、個々のアクターオブジェクトの独立性、アクターごとのメッセージ処理の原子性（逐次性）などで置換することに成功し、稚拙な実装ながらもある程度の性能を披露し、GPU を用いた非同期処理の新たな可能性を提示した [1]。しかし CUDA はメッセージパッシング機構を持たないため、我々が提案した方式を用いても、ユーザはメッセージパッシングのための複雑な記述を要求されることとなった。

本稿ではアクターモデルに基づいた CUDA の実行時システムを利用するための手段として、ActGPU という新たな言語を提案し、そのコンパイラを実装したことについて論じる。ActGPU 言語は Java の似たオブジェクト指向プログラミングで、アクター生成、メッセージパッシングの簡単な実装を可能とし、ActGPU コンパイラは ActGPU コードから、C 言語や C++ の拡張である CUDA C コード（アクター計算の実行時システムコード）への変換を行う。これにより記述性が大幅に向上し、ユーザは GPU で動作するアクターシステムを簡単に開発できるようになった。しかし ActGPU コンパイラが CUDA C への変換を行うのに対して、C++ で利用できる配列などのデータ構造やライブラリ関数などを扱うことができないため、現在は柔軟なプログラミングを行うことが難しく、またコンパイルにおける無駄がまだ多いため、処理性能は前回に劣る結果となった。これに対しては、ActGPU コンパイラの構成を今後変更していくに従い記述の自由度、処理性能は向上していくと考えている。

本稿の構成は以下のとおりである。第 2 節では我々の先の研究である、CUDA でのアクター計算の実行時システムについて簡潔な説明をする。第 3 節では ActGPU 言語

の仕様、コンパイラの構成を論じる。第 4 節では ActGPU を用いたプログラミング例と、その実行を通して ActGPU の評価を行う。第 5 節ではまとめと今後の課題について述べる。第 6 節では本研究の関連研究を紹介する。

2. CUDA におけるアクターフレームワーク

先の研究 [1] にて、CUDA 上で動作するアクターシステムの実装を行った。本項ではその説明を簡潔に行う。

2.1 アクターモデル

最初にアクターモデルについて説明する。アクターモデルは Carl Hewitt [2] に端を発し、William Clinger [3]、Gul Agha [4] などによって完成された、並列環境下における並行計算理論である。アクターシステムは並行に動作する多数の**アクター**と呼ばれる動的オブジェクトによって構成される。アクターは各々の持つ情報を非同期メッセージパッシングによって互いに交換しながら、メッセージの種類に応じた処理（メソッド）を行う。そのため、アクターは自身の状態と、メッセージキューを内包している。並列制御におけるアクターモデルの利点を以下のアクターの特徴から説明する。

- アクター間での情報隠蔽性
- 非同期メッセージパッシング
- 単一アクター内でのメッセージ処理の逐次性

まずアクターは各々の情報をメッセージパッシング以外の方法で知ることができない。これはオブジェクト指向に基づくアクターの重要な性質の一つである。この性質ゆえに、一般的な並列プログラミングにおいて考慮に入れなければならない競合を、ある程度自然に避けることができる。またアクターは他のアクターとの非同期なメッセージ送信や受信を、常に行うことができる。受信したメッセージの処理を開始するタイミングはアクターごとに自由であり、このため並列計算ノードごとの処理量の公平性を保ちやすい。さらに、アクターごとのメッセージ処理は逐次的に行われる。一般にアクターは受け取ったメッセージをメッセージキューに保存し、それを逐次的に実行していく。あるアクターが受信した複数のメッセージを同時並列に処理することはないという意味である。したがって、この点においても競合が発生しないようなモデルとなっている。

以上の特徴よりアクターモデルに基づいたプログラミングは、アクターの生成と、メッセージに対する処理を記述するだけで、スレッド生成や同期などの問題を解決した実装を行うことができる。簡易な記述法がプログラマへの負担を軽くすると共に、理論付けられた並列プログラミングをプログラマに提供する。さらに負荷分散が容易なため性能を向上させやすく、メッセージパッシングという処理が追加されただけであるため、オーバーヘッドが少ない。

実際にアクターモデルに基づく言語はいくつか存在する。

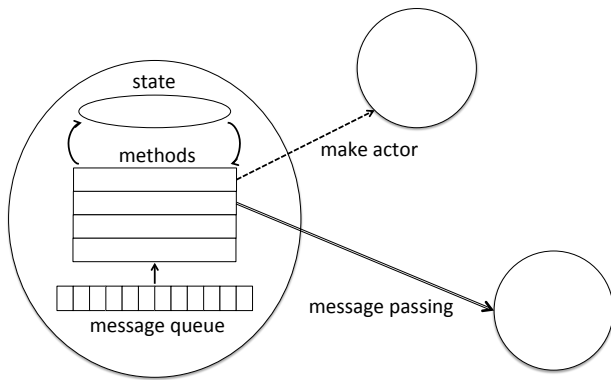


図 1 アクターは受信したメッセージに対応したメソッドの処理によって、自身の状態の更新、新たなアクターの生成、メッセージ送信をおこなう。

近年よく知られるものとして、Scala[5]やErlang[6]などが著名である。この二つはどちらもメッセージパッシング機構を言語レベルで提供しており、各々の文法規則に従った実装を行うことでアクターシステムを動作させることができる。ただし言語によってプログラミング指向は大きく異なる場合も多い。例えばScalaはJavaのクラス定義に似た形でアクターの定義を行う[5]、オブジェクト指向言語である。対してErlangはシステム自体はアクターを用いるが、関数型言語である[6]。また、プログラムを動作させるプラットフォームについても多岐に渡る。マルチコアCPU内での並列や複数CPUでの並列、複数コンピュータ間での並列など、アクターモデルを用いることができる場は多い。なお本研究では、多数の計算ユニットを持つGPU上でアクターシステムを動作させる。

2.2 CUDA 上でのアクター実行時システムの概略

以前に作成したGPGPU上でのアクターシステムは、GPU向けの統合開発環境CUDAで実装を行った。まずはCUDAの計算モデルについて簡単に説明をする。

CUDAではCPUで実行するホスト側のコードと、GPUで実行するデバイス側のコードを記述する[7]。ホスト側とデバイス側で扱うメモリが異なるため、ホスト側でGPU上のメモリ領域確保などを行なっておく必要がある。デバイス側のコードで記述された部分は、マルチコアCPUでの並列処理と同じように、マルチスレッドで処理が行われる。CUDAではマルチコアCPUと違い、GPUの持つ計算ユニットStreaming Processor (SP) よりも大量のスレッドを生成し、SPにスレッドをスケジューリングして並列処理を実行する。なおスレッドの処理は、同じ命令をスレッドごとに違うデータに対して実行する、データ並列な処理である。

次にアクターモデルをCUDAに適応させることを考え

る。そのために、GPUの特徴について考える必要がある。正確な結果を出力し、かつ一定の性能を出すためには、GPUアーキテクチャに従った実装をしなければならない[12]。単にアクターモデルを取り入れただけでは、メッセージパッシング以上のオーバーヘッドが積り、記述性の向上という利得を低速処理という損害が上回ってしまうことになる。CUDAの特徴について抑えなければいけないのは次の二点である。

- メモリアクセスパターン
- SIMD型の命令処理機構

NVIDIA社のGPUには複数種類のメモリが存在する[7]。すべてのスレッドから読み書きできる**グローバルメモリ**、一定数のスレッドを一括りにしたブロック内で共有される**シェアードメモリ**、スレッドごとに備えた**ローカルメモリ**などである。メモリの種類ごとにメモリアクセス速度、記憶容量、キャッシュの有無などに違いがあり、最適なメモリアクセス方法についても特徴がある。CUDAでGPUの性能を最大限に引き出すためには、メモリアクセスパターンを熟慮した実装を行う必要がある。それがCUDAプログラミングの難解な部分の一つである。

また、GPUの多くは根底の部分で *single instruction multiple data (SIMD)* という手法のもとに成り立っている。これはある一つの命令で複数のデータに対する処理を同時に行うことである。例えば、ある変数を自乗する命令を、数値が格納された配列のすべての要素に対して並列に実行するのがSIMDである。NVIDIA社のGPUでは32個のスレッドを一括りにした *warp* という単位を定義し、その中のスレッドに対してはSIMD型の命令処理機構を用いている。if文などで分岐が発生し32個のスレッドが別々の処理をすることになると、命令は一つずつしか処理できないために、分岐先の命令は並列に処理されない。branch divergenceと呼ばれるこの現象を避けるために、warp内では分岐が発生しないようにプログラミングをする必要がある。

我々は以前、他のアクターモデルに基づいた言語と異なりCUDAでアクターシステムを実装するために、以上のGPUの特徴を考慮に入れたシステムを構築した。そのアクターシステムについての説明を行う。まずCUDAに適応させるにあたって、アクターの構造を多少変更した。アクターは自身の状態のみを持ち、メッセージキューはアクターの外部に定義した。代わりにメッセージごとに送信先アクターの場所を保持する。メッセージキューは処理されるべきメッセージの種類だけ用意する。このメッセージ処理の種類とは、アクターごとに定義したメソッドの種類総数である。

全スレッドは同一メッセージキューよりメッセージを取得し、メッセージの送信先アクターを選択、個々のメッセージの処理を行う。メッセージの種類判別はScala[5]と同

じように switch 文によって実装されているが、メッセージの種類ごとにメッセージキューが設けられているため、全スレッドがデータ並列なメッセージ処理を行い、branch divergence を回避することができる。また、スレッドが同一メッセージキューからメッセージを取得する点は、スレッドによる連続メモリ領域へのアクセスも達成できている。メッセージの処理によって新たにアクターやメッセージが生成され、選択するメッセージキューを順番に変更していくことで計算に必要なメッセージ処理をすべて実行し、処理すべきメッセージがなくなった時点で全体の処理が終了する。簡単に説明を行ったが、より詳細な情報については以前の論文 [1] を参照していただきたい。

3. ActGPU

本節では、GPU 上で動作するアクターモデルに基づいた並列言語 *ActGPU* の提案を行う。

3.1 言語仕様

最初に言語の特徴を紹介する。*ActGPU* は単一 GPU 上で動作するアクターモデルに基づいた言語である。独自の文法にしたがったソースコードを、前節で説明した CUDA の実行時システムに変換し、GPU 上でのアクター計算を実行する。CUDA 実行時システムの仕様により、複数のメッセージが発行されると、それらのメッセージが並列に処理される。よってプログラマはアクターと、アクターごとのメッセージ処理内容の定義を行い、同時に複数のメッセージ送信が発生するような記述をすることで、手軽に並列処理を実行できる。

以下に *ActGPU* の言語仕様について説明していく。アクターシステムの実装では、アクターの定義とメッセージ処理内容の記述は必要不可欠である。コード 1 に示すように、*ActGPU* のアクター定義は Java でのクラス定義に似た形で行う。アクター本体には、Java におけるフィールド変数宣言、コンストラクタの定義、メソッド定義を記述する。

ActGPU ではアクターオブジェクトのフィールド変数は個々のアクターの状態変数を表す。現在の仕様では変数の型として扱うことができるのは、int 型と Actor 型であり、他の原子データ型、配列、構造体、オブジェクトなどの複合的なデータ型は利用できない。int 型変数の初期値は定義しない限り -1 である。コンストラクタやメソッドの実行は、メッセージの処理に相当する。*ActGPU* のメッセージパッシングは、メッセージの返り値を期待しない、非同期メッセージパッシングである。このため、メッセージの処理を表す *ActGPU* のメソッドには、通常の言語の関数やメソッドのような返り値という概念はなく、型を定義する必要はない。コンストラクタやメソッドに与える引数は送信されるメッセージが内包した情報を表す。引数に関し

```
1 Actor Act1 {  
2   /* Field declaration */  
3   int n;  
4   Actor a;  
5  
6   /* Constructor */  
7   Act1 (int x, Actor y) {  
8     n = x;  
9     a = y;  
10  }  
11  
12  /* Method */  
13  msg1 (int x) {  
14    Act1 act = new Act1(x, a);  
15    a.msg1(x);  
16    act.msg1(n+x);  
17  }  
18 }
```

コード 1 Actor definition sample

ても現在の仕様では、int 方か Actor 型の変数のみ使用できる。

コンストラクタやメソッド内で記述できるのは、JavaScript の文法にしたがった四則演算や制御文などである。ただし JavaScript で扱うことのできないポインタは利用できず、変数宣言では型として int あるいは Actor 型を指定する。また JavaScript の API 関数なども記述することができない。それらに加えて、アクターシステムのためにアクター生成、メッセージパッシング命令をメソッド内に書くことができる。アクターの生成は、new 演算子を用いてアクターオブジェクトを生成することにあたる。コード 1 の第 14 行目のようなアクター生成式が利用でき、ここでは Act1 の定義に従った act という新たなアクターを生成している。また、メッセージパッシングはメソッドの参照によるメソッド呼び出しで表現される。コード 1 の第 15, 16 行目はメッセージの送信を表しており、特に第 16 行目の命令は Act1 アクターである act に、Act1 で定義された msg1 という処理を行うメッセージを送信することを意味する。なお第 15, 16 行目で送信された二つのメッセージの処理は、後に並列に実行される。

プログラムには特殊なアクターとして Host アクターを記述する。GPGPU における計算は CPU がホスト、GPU がクライアントという立場の分散環境と考えることができる。*ActGPU* においては、クライアント側の計算はアクター群が並列実行することについては述べたが、ホスト側での計算についてもアクターを用いて記述する。このために導入した言語機構が Host アクターである。これは、具体的にはプログラムに対する入出力のためのアクターである。入力のための装置として、Host アクターの main メソッドを実装する。これは本システムの始点に相当するため、main メソッドへの引数はプログラムへの入力値をあらわす。コード 2 の第 3 行目から 6 行目の main メソッドの

```

1 Actor Host {
2   /* main method */
3   main (int n) {
4     Act1 act = new Act1(n, this);
5     act.msg1(1);
6   }
7
8   /* Other methods */
9   print (int n) {
10    printf("%d\n", n);
11  }
12 }

```

コード 2 Host actor definition sample

ように、最初のメッセージパッシング命令を記述することで、GPU 上でのメッセージパッシングが開始する。Host アクター内で定義した main 以外のメソッドは、データの出力を担当する。アクター計算の中で出力したい情報を、メッセージという形で Host アクターへ送信することで、情報の出力がなされる。ただし、C 言語または C++ の文法にしたがった記述をしなければならない。例えば printf などは Host アクターのメソッドに記述することが可能である。処理すべきメッセージがなくなった時点でアクター計算プログラムは自動的に終了するため、終了のための特別な記述をする必要はない。

現在の仕様の大きな筋は以上である。最後に実際のプログラミング過程をまとめる。プログラマはまず計算に必要なアクターの定義、メッセージの処理としてのメソッドを記述する。そして Host アクターを定義し、main メソッドに計算の始点となるアクター生成を、それ以外のメソッドに外部出力コードを記述する。完成したプログラムをコンパイルし、生成された CUDA コードファイルを CUDA 実行環境におき nvcc でもう一度コンパイルし実行ファイルを生成、最後に実行ファイルに Host アクターの main メソッドに定義した引数を与えることで、作成したプログラムを実行することができる。

3.2 コンパイラの構成

ActGPU のコンパイラは前述したように、ActGPU 言語で書かれたコードを CUDA のコードに変換して、既存の CUDA コンパイラである nvcc (NVIDIA CUDA Compiler) でもう一度コンパイル、実行ファイルを出力する。本研究で開発を行ったのは ActGPU コードから CUDA コードへの変換器である。

まず、コード変換の概要を表す。図 2 のように、ActGPU コードから CUDA コードへの変換は、3 ステップを経て行われる。

- (1) ActGPU コードから JSON (ActGPU) への変換
- (2) JSON (ActGPU) から JSON (CUDA) への変換
- (3) JSON (CUDA) から CUDA コードへの変換と、変換

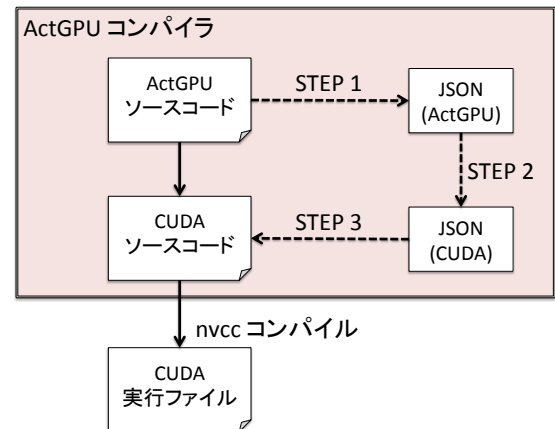


図 2 我々が作成した ActGPU コードから CUDA コードへの変換器は 3 ステップに渡る変換を行う。

済みコードのテンプレートへの埋め込み

ActGPU コードを理解しやすいデータ形式へ変換、データの整理を行い、最後に CUDA で動作する目的コードを生成する..

最初に ActGPU コードから JSON (ActGPU) への変換について説明する。JSON[9] は JavaScript Object Notation の略称で、テキストベースの軽量なデータ記述言語である。これを用いる理由は、JSON の形式が直感的に理解しやすく解析が容易であること、また David Majda の PEG.js[11] を利用するためである。PEG.js は JavaScript で書かれた、**解析表現文法** (Parsing expression grammar, PEG)[10] に基づいたパーサジェネレータである。本稿では詳しくは説明しないが、分析的形式文法 PEG に従って対象言語の解析規則を構築すると、対象言語向けのパーサを自動で生成できる。本研究では Majda による、JavaScript コードを解析して JSON に変換するサンプルコード javascript.pegjs に新たな規則を追加することで、ActGPU 言語向けのパーサを生成するパーサジェネレーターを作成した。ActGPU の基本的な部分が JavaScript に準拠しているのは、規則の追加という形をとっているため既存部分は JavaScript の規則にしたがうことに起因する。追加した規則は具体的に、アクター生成とアクター内のメソッド定義の部分、そして変数宣言の部分である。アクター生成とメソッド定義については、それらがもともと JavaScript には存在しないため、変数宣言については int や Actor 型で宣言できるようにするため新たな規則を定義した。追加分である pegjs ファイルの内容を本稿末尾に掲載する (A.1)。ActGPU の文法について詳しく理解するために javascript.pegjs[11] とともに参照していただきたい。

次に JSON (ActGPU) から JSON (CUDA) への変換について説明する。まず第一に今回のコンパイラはコードの最適化を含まない。我々が以前発表したアクターモデルに基づいた CUDA での実行時システムのコードを、実行可

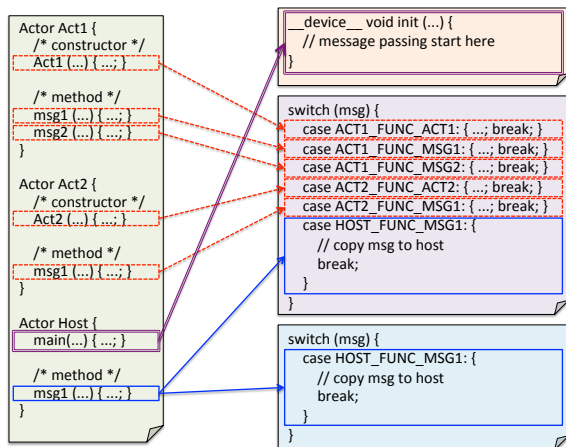


図 3 個々のメッセージの処理を switch 文のパターンに変換する。Host アクターのメソッドについては特殊な変換を行う。

能な形で出力するのみである。将来的にはコンパイラ部分で最適なメモリアクセスを実行するような CUDA コードを生成したいと考えているが、今回の発表時点ではできておらず、今後の課題の一つとなっている。しかし、最適化は行わないにせよ、ActGPU コードと生成されるべき CUDA コードは大きく異なっている。そこでデータの整理を含めた上で、一度 JSON から JSON への変換を行うことにした。ActGPU の JSON から直接 CUDA コードを生成することも考えたが、この場合 ActGPU の文法を変更したり CUDA でのアクターシステムの処理方法が変更されるごとに一から逆変換器を生成しなければならない。JSON から JSON への変換の方は直接変換より容易であり、CUDA コードの JSON 形式を決定してそれに対応した逆変換器を作成しておけば、逆変換器の変更はほとんど必要なくなるため、開発効率が上がると判断した。よって、このステップはコンパイラの処理の核となる部分であるといえる。

変換の全体方針は、定義されたコンストラクタやメソッドの処理を、メッセージ受信ハンドラである switch 文のパターンに変換することである。ただし、前述したように Host アクターは CPU 側の処理をアクターで記述したものである。そのため Host アクターは特殊な扱いが必要であり、他のアクターとは別の方針に従った変換規則を設定した。図 3 にあるように、Host 以外のアクターのメソッドは GPU 上で処理するコードとして変換したが、Host アクターの main メソッド以外は CPU 上で処理するコードとして変換を行った。具体的には、CPU でも GPU での処理と同じように switch 文を用いることで、Host アクターのメソッドに対応するメッセージの処理を実行できる形にした（ただし CPU でのメッセージ処理は逐次実行である）。

言語仕様の説明において Host アクターは入出力のためのアクターであると述べたが、この変換規則を用いることで図 4 に示すような全体の処理の流れを構成した。最初に

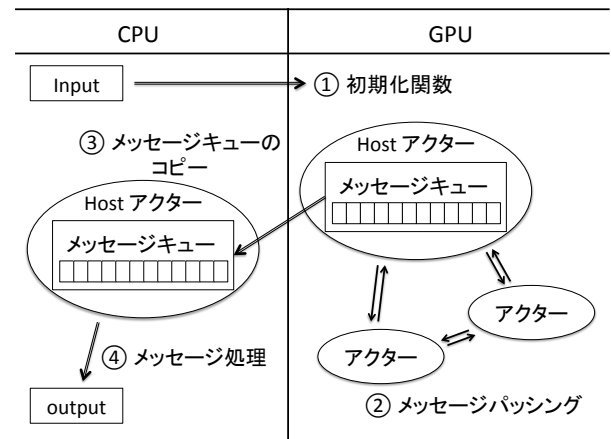


図 4 生成された CUDA コードの処理の概要。GPU カーネル関数が呼ばれるのは一回のみであり、すべてのメッセージ処理が終わるまで CPU に処理は帰らない。

CPU 側でプログラム実行のための入力受ける。次に入力を GPU 側のメモリへコピーし、Host アクターの main メソッドを変換した初期化関数を実行する。この関数の中で最初のメッセージパッシングを実行することで、アクター間でのメッセージパッシングが連鎖的に発生し、並列に計算が進む。基本的にはアクター内に記述したメソッドの処理を実行するが、Host アクターにメッセージを送信した時のみ Host アクターのメッセージキューへメッセージを溜める。やがて処理すべきメッセージがなくなったところで、Host アクターのメッセージキューの内容を CPU 側にメモリコピーする。最後に CPU のコードとして記述された、Host アクターの main 以外のメソッドの処理を実行し、必要なデータの出力を行う。

CPU 側でのメッセージ処理の実行は、先の実行時システムの段階では実装していなかった。CUDA C を用いているため、この CPU での処理は C 言語または C++ で記述される。Host アクターの main メソッド以外は C 言語または C++ で書くという言語仕様は、ここでの自然な変換のためである。

具体的な変換の中では特に、アクター生成、メッセージ送信、変数への代入などの命令、そしてコード中での変数のスコープ部分で注意が必要となる。ActGPU のアクター生成は CUDA ではアクター生成、アクターの状態の初期化、生成したアクターへのメッセージ送信（該当するメッセージキューの選択、メモリへの書き込み）にあたる。ActGPU のメッセージ送信は CUDA では該当するメッセージキューの選択、メモリへの書き込みにあたる。変数への代入は、特にアクターの状態の更新の場合に、アクターへの変数代入コードに当たる。変数スコープは三種類のスコープがあることを考える。

- アクターの状態
- メッセージのデータ

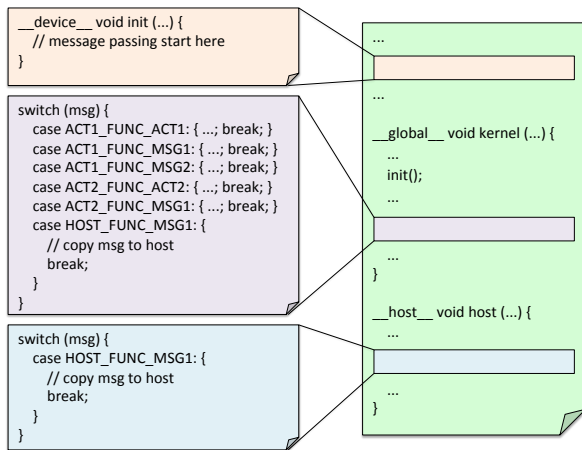


図 5 生成した switch 文などをテンプレートに挿入することで CUDA コードが完成する。

- 一時変数

変数がアクターの状態やメッセージのデータに該当する場合は、それらを参照できるようにコード変換をする。なお他の部分（基本的に JavaScript の文法に従って記述された部分）に関しては、特に変換をしなかった。このため変換後の JSON (CUDA) の多くの部分は JavaScript コードから変換した JSON (JavaScript) と似通っており、それに揃える形でアクターモデルに関与した部分も JavaScript の文法に対応する JSON 形式に変更した。

最後に JSON (CUDA) から CUDA コードへの変換である。この部分は単純な変換であるため、特に説明をしない。前述したように、JavaScript コードを変換したような形となっているため、逆変換器は JavaScript を対象としたものを多少変更するだけでよい。ただ本項の最初に記したように、実際は JSON から CUDA コードへの逆解析だけでなく、逆解析によって生成されたコードをテンプレートに埋め込む形で最終的なコードを生成している。JSON から JSON への変換は、前述したように switch 文への変換と、最初に実行される関数、CPU 上でのメッセージ処理の switch 文の生成にあたる。スレッドごとの動作や全体の処理の流れは実行時システムと同じであるため、処理の枠組みやデバイスメモリ領域確保命令などは前回のサンプルプログラムを参考にテンプレートとしてまとめた。現在はアクターやメッセージキューの確保領域量、計算に使用するスレッドの総数などはテンプレートに記述されており、ActGPU コード内では定義できない。この部分については今後変更を加えていくつもりである。

なお CUDA コードへの変換を行うが、CUDA コードにおける計算に利用するスレッドやブロックの数、アクターやメッセージのための GPU メモリ領域の確保を行うためのコードは、コンパイラがサポートしていないため ActGPU 言語では記述できない。それぞれ任意に設定してあるが、変更するには ActGPU コンパイラによって生

```

1 Actor Test {
2     Test(int n, Actor a) {
3         a.print(n);
4     }
5 }
6
7 Actor Host {
8     main(int n) {
9         for (int i = 0; i < 1280; i++) {
10             new Test(n+i, this);
11         }
12     }
13
14     print(int n) {
15         printf("%d\n", n);
16     }
17 }

```

コード 3 Sample code

成された CUDA ファイルを直接編集しなければならない。ActGPU コード上で編集する機能に関しても今後追加していく予定である。

4. システム評価

本節では幾つかの ActGPU コードサンプルを示し、コンパイル、実行することで ActGPU コンパイラの評価を行う。

4.1 簡単なサンプル

まずは簡単なサンプルについて紹介する。

コード 3 は 1280 個の Test アクターを最初に生成、メッセージパッシングを行い、そのメッセージに対応するコンストラクタの処理によって、Host アクターの print 処理を行うメッセージ送信が発生、受け取った値を表示するプログラムである。入力値 `n` を用いて main メソッドの内容を実行し、生成された Test アクターのコンストラクタ処理が GPU 上で並列に処理される。GPU カーネル関数に `n` を与えて実行するだけなので、ここではメモリコピーは発生しない。Host アクターの main メソッドや、print メソッドはコンパイラの仕様により逐次処理となる。Test コンストラクタで送信する Host アクターの print メソッドに従うメッセージは Host アクターのキューに入れられ、並列に処理されていた Test コンストラクタの処理が終了した時点で、キューに溜まったメッセージの CPU メモリへのメモリコピーが実行される。最後に Host アクターの print メソッドの内容が CPU 上で実行され、それぞれの Test アクターより受信したメッセージデータを表示する。

ほとんどメッセージパッシングの意味をなさない例であるが、このように Host 以外のアクターのメソッド呼び出しという形で並列処理を行うことができる。メソッド呼び出しは Java や C 言語でもよく利用する記述であるため、実装も簡単であるといえる。

```

1 Actor Fib {
2   Fib(int n, int slot, Actor r) {
3     if (n < 2) r.value(slot, n);
4     else {
5       Add a = new Add(slot, r);
6       Fib f1 = new Fib(n - 1, 0, a);
7       Fib f2 = new Fib(n - 2, 1, a);
8     }
9   }
10 }
11
12 Actor Add {
13   int slot;
14   Actor r;
15   int n0 = -1, n1 = -1;
16
17   Add(int slot, Actor r) {
18     this.slot = slot;
19     this.r = r;
20   }
21
22   value(int slot, int n) {
23     bool answer = false;
24     if (slot == 0) {
25       n0 = n;
26       answer = n1 >= 0;
27     } else {
28       n1 = n;
29       answer = n0 >= 0;
30     }
31     if (answer)
32       r.value(this.slot, n0 + n1);
33   }
34 }
35
36 Actor Host {
37   main(int n) {
38     new Fib(n, 0, this);
39   }
40
41   value(int slot, int n) {
42     printf("%d\n", n);
43   }
44 }

```

コード 4 Example code, fibonacci

なおコンパイラの構成の最後に説明したが、現在計算に利用するスレッドの数の定義やメッセージキューのための領域確保はこのコード中に記述することができないため、それぞれ任意に実行されている。メモリ領域に関しては十分な量を確保してあるが、アクターを増やすと実行できない場合がある。前述したように、これらを変更するには CUDA ファイルを編集する必要がある。

4.2 フィボナッチ数計算プログラム

実際の利用例として、ActGPU 言語でフィボナッチ数を計算するプログラムを実装した。

Host アクターの main メソッドで、メッセージパッシングの起点となるアクターを生成することで計算を開始す

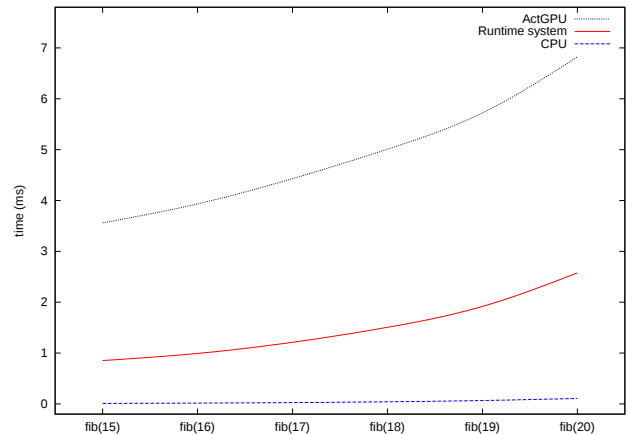


図 6 フィボナッチ数計算における性能比較

る。メッセージパッシングにより、Fib アクターは新たなアクターを動的に生成し、再帰によって木構造を構成する。木構造の葉まで計算が終わると、総和を求めるために Add アクターへのメッセージ送信を開始する。Add アクターは 2 つの値をメッセージで受け取り、自身の親ノードへ足した値をメッセージ送信する。最後に Host アクターの value メソッドに対応するメッセージが Host アクターのメッセージキューへ送信され、GPU から CPU へのメモリコピーの後に計算結果を出力する。

フィボナッチ数を計算するプログラムは、以前の実行時システムでも実装をした。今回の ActGPU 言語を用いたプログラミングで、前回の手書きのコードと同等の品質のコード生成ができれば上々の結果といえる。前回との比較として処理時間の差を図 6 に表す。なお GPU は NVIDIA Tesla C2075 一つを用い、ActGPU と実行時システムはどちらもブロック数 42、ブロックあたりのスレッド数 128 の環境で処理を行った。CPU は Intel Xeon E5645 を単一コアで、C 言語で実装したプログラムを動かした。

比較したところ、処理速度は大きく劣る結果となってしまった。以前の実装ではメッセージの種類が二種類であったのに対し、今回は四種類あることがこの結果の最大の原因であると考えられる。GPU 上の処理において、スレッドへのタスクスケジューラはメッセージキューを順番に選択し、選ばれたキューのメッセージに対応する処理をスレッドに実行させる。そのため、メッセージの種類が二倍になったことでメッセージキュー選択の回数が二倍の回数となり、メモリアクセスの回数も増加している。また、生成された CUDA コードは、安定な動作を保証するために無駄なメモリアクセスを行なっている。グローバルメモリに置かれたアクターやメッセージへのメモリアクセスの際、一度アクセスした値をレジスタに保存せず必要となったときに毎回グローバルメモリまでアクセスする方式であるため、大きなオーバーヘッドが生まれている。これらは変換によってコードの最適化を実行しない現在のコンパイラの

仕様では避けることのできないオーバーヘッドである。ただし、どちらもコンパイラの仕様の見直しによる改善は可能である。前者は定義したメソッドすべてをメッセージの処理に変換せずに、定義されるメッセージの種類を制御するような変換方式に変更し、後者は変数呼び出しを記録することで対応できると考えている。

記述性に関しては、かなり直観的な記述ができるようになったといえるであろう。フィボナッチ数の計算プログラムを先に表した簡単なプログラムと比較すると、動的なアクター生成、再帰的なメッセージパッシングなどに相違点が存在するが、ActGPU 言語では new 演算子を用いたアクターオブジェクトを生成、または同一メソッドの呼び出しをメソッド内で定義するだけなので、実装は難しくなく、また、同期や排他制御のための特別な記述なしに並列処理を実装できている。なお、実行時システムのコードが約 240 行であるのに対し、図??に示されるように、ActGPU 言語では 40 行程度の記述によって実装されており、記述力の向上を明確に感じることができる。

以上より、コンパイラによるコード変換の最適化を模索する必要があるといえる。愚直に変換するのではなく、メソッド内で何度も利用される引数などがあった場合、レジスタを利用するようなコード変換にするだけでも高速化できる。

5. まとめ

アクターモデルに基づいた、GPU 上で動作する非同期並列プログラミング言語 ActGPU のコンパイラを作成した。ActGPU 言語は Java ライクな言語であり、直観的な記述によって GPU を用いた並列処理を記述できるようになった。反面処理性能に関しては、CUDA でアクターシステムを直接実装したものに大きく劣る結果となった。これは現在の ActGPU コンパイラがコードの最適化をしないことが原因であるといえる。それを含めて今後の課題は多い。

第一の課題は、GPU の実行コードの最適化によって処理性能を向上させることである。メモリアクセスパターンやスケジューリング方式の変更によって、高速化できる部分は多い。Che ら [13] はメモリ上のデータを再配置することで最適なメモリアクセスに誘導しているが、それを参考にアクターやメッセージの読み込み、書き込みの際にデータの再配置をするアルゴリズムを実装することを考えている。また、現在はメッセージの種類数に応じたメッセージキューを用意することで branch divergence を回避しているが、キューに存在するメッセージの数が一つでも数千個のスレッドでメッセージキューにアクセスする点や、生成されるメッセージが少ない場合メッセージキューの占めるメモリ領域が無駄になってしまうなどデメリットが存在する。Fung ら [14] や Han ら [15] のように、スレッドへのタスクスケジューリングを制御する形で warp の処理を揃

え branch divergence を回避する方法への転換する方法を参考にしたい。それに関連して、CUDA のスレッドスケジューリングに *Lazy Task Creation (LTC)* [16] を取り入れることも思索している。LTC の構想を CUDA に導入することから始めなければならないが、理論付けられたスケジューリング方式を取り入れることのメリットは大きい。これらの最適化をコンパイルの時点で実行することも考えており、今回提案したコンパイラの構成は今後大きく変わる可能性もある。

第二の課題はユーザの自由度を高めていくことである。まず ActGPU の機能を増やす必要がある。現在は配列も用いることができないため、実装できない計算も多い。また CUDA のスレッドやブロックの数、確保するメモリ領域のサイズなどをユーザが自由に指定できるようにすることで、並列度の向上や、無駄なメモリ領域の消費の解消などをユーザに委ねる形にしたい。最適なスレッド数などは実行したいプログラムによるところも大きいので、ユーザが設定したほうが都合が良い場合が多いだろう。

第三の課題として、より大きな問題に対応するためにシステムのマルチ GPU 化を検討している。これに関しては今回の実装のアイデアをうまく用いることができそうである。アクターやメッセージのデータ出力のために、一度 CPU 側にメッセージをコピーしたが、これは複数 GPU 間でのメッセージのやりとりに応用できる。コピーしたメッセージの処理を CPU で行わずに、別の GPU のメモリにもう一度コピーすることで、GPU 間でのメッセージパッシングが可能となる。

6. 関連研究

本稿で何度も名前をあげた、Scala[5] や Erlang[6] などにはアクターモデルに基づいた言語であるという点で関連している。特に Scala のメッセージパッシングは本研究で大いに参考にした。これらのアクターモデルに基づく言語は多々存在するが、本研究との大きな相違点は処理を行うプラットフォームである。マルチ CPU や分散環境で利用できる言語に対し、我々は GPU でアクターシステムを利用できる言語を作成した。第 2 節でも述べたように GPU は一般的な並列・分散環境にない特徴を持っており、それに合致した実装をしているため、他の研究と異なっている。

GPU 上でアクターシステムを利用するコンセプトの研究も存在する。Chapel というあらゆる並列環境に適応できる言語にアクターモデルを実装させる研究 [17] では、GPU 環境でアクターシステムを動作させることも構想に含んでいる。しかしその研究の中心は、Chapel にアクターモデルを取り入れることであり、GPU での処理の最適化などは Chapel の機能にまかしている点で我々の研究と違う。本研究は GPU での処理を中心として考えており、そのためメモリアクセスの最適化など、処理の高速化を目指すこと

も含んでいる。

本研究の独自性はアクター計算を実行する環境に GPU を用いている点であり, GPGPU コンピューティングの新たな可能性を与えるという点で貢献していると考えている。

参考文献

- [1] 高柳亘, 脇田建: CUDA を用いたメッセージ送信型並行計算 Actor モデルの実装, HPC-135 (2012).
- [2] Hewitt, C., Bishop, P., Steiger, R.: *A universal modular actor formalism for artificial intelligence*, IJCAI 73 (1973), pp 235-245.
- [3] Clinger, W.D.: *Foundations of Actor Semantics*, MIT (1981).
- [4] Agha, G.A.: *ACTORS: a model of concurrent computation in distributed systems*, The MIT Press (1986).
- [5] Haller, P.: *Scala Actors: A Short Tutorial* (online), <http://www.scala-lang.org/node/242> (2012.11.06).
- [6] Armstrong, J.: *Programming Erlang: Software for a Concurrent World*, Pragmatic Bookshelf (2007).
- [7] NVIDIA : NVIDIA CUDA C Programming Guide Version 4.2 (2012).
- [8] KHRONOS : OpenCL Overview, <http://www.khronos.org/opencl/> (2012.11.06).
- [9] Introducing JSON (online), <http://www.json.org/> (2012.11.06).
- [10] Bryan, F.: *Parsing Expression Grammars: A Recognition-Based Syntactic Foundation*, POPL 2004 (2004)
- [11] PEG.js: Parser Generator for JavaScript, <https://github.com/dmaji/pegjs> (2012.11.06).
- [12] Nickolls, J. Buck, I. Garland, M. Skadron, K.: *Scalable Parallel Programming with CUDA*, Queue-GPU Computing Vol 6 (2008), pp 40-53
- [13] Che, S. Sheaffer, J.W. Skadron, K.: *Dymaxion: Optimizing Memory Access Patterns for Heterogeneous Systems*, sc11 (2011)
- [14] Fung, W.L. Sham, I. Yuan, G. Aamodt, T.M.: *Dynamic Warp Formation and Scheduling for Efficient GPU Control Flow*, MICRO 40 (2007), pp 407-420
- [15] Han, T.D. Abdelrahman, T.S.: *Reducing branch divergence in GPU programs*, GPGPU-4 (2011)
- [16] Mohr, E. Kranz, D.A. Halstead, R.H.Jr.: *Lazy task creation: a technique for increasing the granularity of parallel programs* LFP 90 (1990)
- [17] Shali, A.: *Actor Oriented Programming in Chapel*, University of Texas (2010)

付 録

A.1 ActGPU 解析文法

```

1 VariableStatement
2   = TypeSpecifier
3   VariableDeclarationList EOS
4
5 TypeSpecifier
6   = VarToken / ActorToken / Identifier
7
8 ActorToken
9   = "Actor"
```

```

10
11 VarToken
12   = "int"
13
14 ActorDeclaration
15   = ActorToken Identifier
16   "{" ActorBody? "}"
17
18 ActorBody
19   = (ActorElement)*
20
21 ActorElement
22   = Statement
23   / MethodDeclaration
24
25 MethodDeclaration
26   = Identifier "(" MethodParaList? ")"
27   "{" FunctionBody "}"
28
29 MethodParaList
30   = TypeIdentifier ("," TypeIdentifier)*
31
32 TypeIdentifier
33   = TypeSpecifier Identifier
```
