

CUDA を用いたメッセージ送信型並行計算 Actor モデルの実装

高柳 亘¹ 脇田 建^{1,2}

概要：アクター計算は非同期メッセージ通信に基づいた並行オブジェクト指向計算モデルであり、動的、非定型な並列処理の記述に優れている。本稿では、CUDA C を用いてアクター計算の実行時システムを NVIDIA Tesla C2075 に実装した経験に基づき、その実証手法について報告する。評価実験において、アクター計算における非同期メッセージ通信、メッセージ処理の原子性、動的なアクターとメッセージの生成などが正しく動作できたことを確認した。また、ノード数 16,384 個のsmall-worldネットワーク上の最短路計算において単体 CPU に比べて 8.5 倍の高速化に成功した。

キーワード：アクター計算、メッセージ通信、CUDA

An implementation of the Actors message-passing concurrent computation model using CUDA

WATARU TAKAYANAGI¹ KEN WAKITA^{1,2}

Abstract: Actor computation is an concurrent object-oriented programming model that is based on asynchronous message passing and can describe dynamic and irregular computation patterns. The article proposes an implementation scheme of Actor computation on an NVIDIA Tesla C2075, using CUDA C. Through a few Actor-based applications, all the features that Actors provide, such as asynchronous messaging, atomic processing of messages, and dynamic creation of actors and messages, have been examined. The Actor-based parallel computation of shortest path length problem over a small-world network that involves 16,384 nodes exhibited 8.5 speed-up in comparison with a CPU core.

Keywords: Actor model, message passing, CUDA

1. はじめに

近年爆発的に増加する情報をどう扱うかが問題となっている。身近にもインターネットという巨大な情報網が存在し、現代は情報爆発時代と呼べる。

大規模データを扱うための計算資源の性能は順調に向上している。理化学研究所の『京』や、東京工業大学の『TSUBAME』などスーパーコンピュータとよばれる大規

模高速計算機は最先端の計算資源である。昨今専門家でなくともスーパーコンピュータを利用できるような環境が整ってきている。

計算資源の成長が著しい中で、並列処理や分散処理など資源を有効活用するための方法の研究も盛んである。しかし逐次処理では考慮しない同期や排他制御を組み込む必要性があることが、プログラマに対して高い障壁となっている。それらは、並列処理や分散処理で矛盾を起こさないプログラムを設計するためには必須の技術である。

並列・分散処理を容易に実装するための計算モデルの研究は古くから行われてきており、近年その重要性も高まってきている。

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

並列プログラミングのための計算モデルとしてアクター計算 [1] が著名である。アクター計算は情報隠蔽されたオブジェクトがメッセージを介して非同期的な処理を行う並行オブジェクト計算モデルの一種である。アクターに基づいた並列プログラミング言語としては ABCL/1[3] が知られ、Erlang[4]、Scala[5] など先進的な汎用プログラミング言語にもその機能が取り込まれ、Twitter の高速なメッセージ交換に Scala のアクター機能が応用されるなど、近年、注目が集まっている。既存の言語環境にアクターを取り入れるような試み [6] もある。

本研究の目的はアクター計算を GPU 上で実現することで、柔軟で簡易かつ、安価な並列プログラミングの基盤を提供することである。アクター計算を GPU に実装する上での困難は、動的かつ非同期的なアクター計算の計算モデルを正則的な計算を同期的に実行する GPU の動作モデルに如何に効率的に写像するかという点にある。本研究ではメッセージの種別ごとに設けられたメッセージキューに保存されたメッセージ群のスケジュールを中心にスレッドのスケジュールを行うことで、場合によっては性能劣化の原因となる branch divergence の問題を避ける方針を取った。さらに軽量な排他制御機構の採用、同期コードの除去、不要な初期化の除去などの効率化を施すことによって処理の効率化を図った。

アクター計算の機能を網羅的に使用する並列フィボナッチの例題によって、アクターの動的生成、非同期のメッセージ送信、メッセージ処理の原子性の保証、情報隠蔽などのアクター計算の本質が提案した実装方式で実現されることを確認した。また、スモールワールドネットワークに対する一対全最短距離問題の実装においては単体の CPU に対して 8.5 倍の高速化に成功した。アクター計算を実施するハードウェアとして GPU は一見すると全く不適合にも思えるが、今回の結果は、提案した実装が未だ技術的に稚拙な段階にあることを考え合せれば GPU を用いた非同期処理の可能性を示唆するものと考えられる。

本稿の構成は以下のとおりである。第 2 節では Gul Agha のアクターモデルの説明をする。第 3 節ではアクターを用いた実行時システムの実装について論じる。第 4 節では非同期処理を安全に実行するために、本研究で使ったテクニックについて述べる。さらに高速化についても論じる。第 5 節では上記した例を用いて本システムの評価を行う。第 6 節ではまとめと今後の課題について述べる。

2. Actor モデル

Gul Agha が提唱した**アクター計算**という並行計算モデルでは、メッセージを介して通信する**アクター**と呼ばれる並行オブジェクトが計算を担う。

個々のアクターは状態変数に固有の状態を保持する。状態変数には他のアクターへの参照や値を記録することが

できる。アクターの状態は**情報隠蔽**されており、他のアクターの状態変数を直接、参照することはできない。

相互に情報隠蔽されたアクター間の相互作用は**メッセージ通信**を介して間接的に達成される。メッセージは、受信すべきアクターへの参照、そのアクターでの処理内容を表す**メッセージ名**、引数の値から構成される。

各アクターは受信したメッセージごとに、メッセージ名に対応した処理を原子的に行う。この結果、通信は並行的に行われるが、アクター内に並行性はない。この性質を**メッセージ処理の原子性**という。

メッセージの処理は以下のような操作を含む。

- 状態変数への代入による状態の更新
- 高々有限個の新規アクターの生成
- 高々有限個のメッセージの送信

アクターはメッセージを非同期的に送信するが、メッセージ処理においてはメッセージの送信順は保証しない。このことは各アクターがメッセージキューを持つと説明されることが多いが、正確にはアクターに届いたメッセージ群は集合的に扱われ、メッセージ群から無作為に選択されたメッセージが処理される。

アクター計算は形式的な計算モデルであり、アクターやメッセージの数に制限を設けない。このため潜在的に無限に並列な世界を表現できる。しかしながら、アクター計算に基づいた並列プログラミング言語を実装するには、計算資源に応じた実装上の工夫が求められる。

アクター計算が各種の並行プログラミング言語に取り入れられた背景には、この計算モデルの抽象化を通して並行計算における複雑性が単純化され、平易に理解できる点がある。たとえば、アクター計算は多数のアクターが独立に実行することで高度の並列性を記述できる一方、メッセージ処理の原子性によりアクター内の並列性が排除されている。また、アクターの情報隠蔽機能は、メッセージ処理のみならず、データについてもアクターごとに局所化されていることを意味する。アクター計算を採用することにより、複雑になりがちな並列計算を局所的、かつ逐次的な計算の組み合わせとして理解できるため、並列プログラムの理解は容易になる。

さらに、本稿が議論する GPGPU を含めて、多くの並列計算機は線形代数に代表される定型的な処理の記述は容易だが、動的データ構造に対する非定型的な処理の記述は難しいことが多い。一方、アクター計算においては、アクターの相互参照によって構成した動的構造を再帰的なメッセージ通信によって動的に渡り歩くことができるため、動的な計算の記述には特に有効である。

3. 実行時システムの実装の概要

本研究では前節で紹介したアクター計算に基づいた並列プログラミング機構の実行時システムを、CUDA を用いて

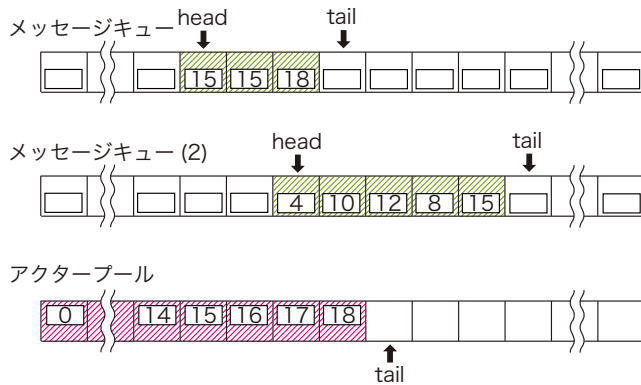


図 1 アクタープールとメッセージキュー

GPU 上で実装することを一つの目標としている。一般的なアクター計算では、アクターの構造や総数、メッセージの数に制限は設けられていない。しかし、現実には計算資源が限られる GPU で効率的に並列計算するためには、アクター計算のモデルに適切に制限を加えることが求められる。本節では、そのような制約を含めた実装方式を概説する。

3.1 アーキテクチャ

アクター計算を表現するための主要な構造は、システムに存在するアクター群とメッセージ群の表現と管理方法である。提案する実装方式において、各アクターの状態変数の容量を定数として扱い、アクター群を**アクタープール**と呼ぶ固定長の配列に保管することとし、新たにアクターが生れるときは、ここに追加することとした。(図 1 参照) アクタープールは、GPU の計算が開始するときに初期化され、アクター生成時にアクタープールに余分がない場合は、ヒープオーバーフローとして扱う。

メッセージ群は、メッセージ名ごとに個別の**メッセージキュー**と呼ぶ固定長の配列領域を用意して管理する。各メッセージは、メッセージの送信先、引数の列を有するが、メッセージを固定長で扱うために、引数の個数は定数に制限した。メッセージキューは二つのポインタ *head* と *tail* を用いた有限バッファによって表現している。新たなメッセージはメッセージキューの *tail* 側に追加し、メッセージキューからメッセージを取り出す場合には *head* 側から得る。

3.2 同期的ハードウェアの上での非同期計算の実現

CUDA の実行モデルは、同期的なマルチスレッディングである。一方、アクターの計算モデルでは多種多様なアクターが異なるメッセージを非同期的に実行する。本研究における基本的な困難は、アクター計算における異種混合非同期的計算をハードウェアレベルの一樣同期的計算にきれいに写像することにある。最初の工夫として、すでに述べたようにアクター計算全体のメッセージ群をメッセージ名に応じて異なるメッセージキューに分割した。同期

的に実行するスレッドに等しく同じメッセージキューからメッセージを与えることによって、GPU 内の全スレッドが同種のメッセージを処理することとなる。

図 2-(a) は GPU のハードウェアスレッドがメッセージキューから取得したメッセージを処理の様子を表している。各スレッドは、メッセージキューの *head* からスレッド番号をオフセットとした位置よりメッセージを取得する。ここで各スレッドはメッセージキューの異なるスロットをアクセスするために同期は不要である。メッセージには、その宛先のアクターへの参照が保存されている。図 2-(a) では、最初の三つのスレッドが得たメッセージはそれぞれ、アクタープールの 15 番、15 番、18 番スロットに保存されたアクターへ送られたものである。メッセージキューに十分なメッセージがない場合には、スレッドは *tail* より右の位置の空のメッセージを受け取ることとなる。

空でないメッセージを受け取ったスレッドは、メッセージの宛先に指定されるアクターにおいてメッセージの処理を行おうとする。ただし、メッセージ処理の原子性を保証するために、複数のスレッドが同一のアクターでメッセージ処理することは避けなくてはならない。この原子性は、次節に述べるように `atomicCAS` 命令を用いた排他制御によって実現した。図 2-(a) では最初の二つのスレッドが得たメッセージがともに第 15 番のアクターに宛てられたことを表しているが、メッセージ処理の原子性により二番目のスレッドが実行権を得て、最初のスレッドがアクターの取得に失敗した様子を描いている。

メッセージの宛先のアクターへのアクセス権限を得たスレッドは、宛先のアクターにおいてメッセージ名に対応した処理を実行し、他のスレッドは休止する。アクター計算と同様に、メッセージの処理においては、計算を行いその結果をアクターの状態変数に保存し、新たなアクターやメッセージを生成することができる。このうち、状態変数の更新については、メッセージ処理の原子性によってアクセス競合が発生しないことが保証されている。一方、残りの処理の実装はアクタープールやメッセージキューのような大域的な構造の操作を伴うために、注意を要する。本実装においては、メッセージ処理中に生れるアクターやメッセージはスレッドごとに局所的な領域に作成し、全スレッドでのメッセージ処理の完了を待つ。

本提案のメッセージ処理においてスレッドは、空のメッセージを得るか、メッセージを得たもののアクターへのアクセス権限を得られないか、無事にメッセージ処理を完了しているかのいずれかである。メッセージ処理の最後のステップにおいて、メッセージ処理を完了したスレッドは、メッセージ処理中に作成したアクターやメッセージをアクタープールやメッセージキューにコピーする。また、アクターへのアクセス権を得られなかったスレッドは取得した未処理のメッセージをメッセージキューの *tail* 側に追加す

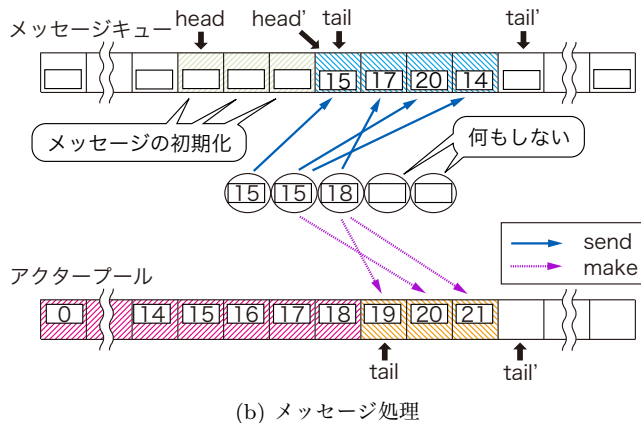
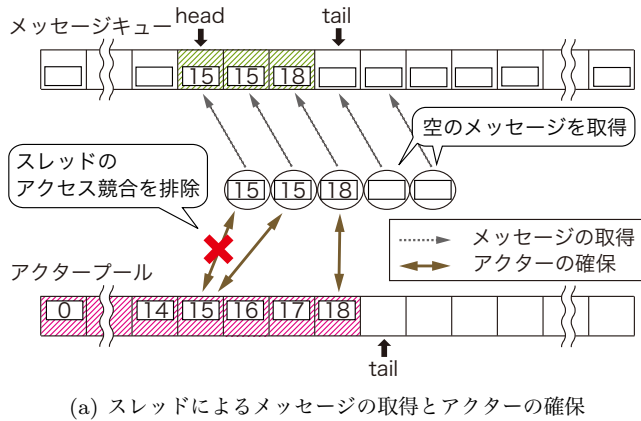


図 2 本システムの処理の 1 サイクル

る。この方式は、メッセージ送信の順序を保証しないアクター計算の性質を上手に活用したものである。

図 2-(b) において、最初のスレッドは処理できなかった第 15 番アクターへのメッセージを書き戻し、2 番スレッドはメッセージ処理中に第 20 番、第 14 番アクターへのメッセージをメッセージキューに追加し、新たなアクターを第 20 番アクターとしてアクタープールに生成している。

すべてのスレッドのメッセージ処理が完了した時点で、メッセージキューのなかで処理済みのメッセージが保存されていた領域は空メッセージとして初期化する。アクタープールとメッセージキューの tail 値は即座に更新され head 値は後で更新される。

3.3 メッセージ処理の実装

前節で述べた処理の流れを疑似コードで表したものが図 3 である。

アクタープールとメッセージキューの管理には構造体を用いる。アクタープール (actors) はプールを表すバッファのほかに tail をフィールドに持つ。メッセージキュー (messages) はキューを表す有限バッファのほかに head, tail, そしてバッファの大きさをフィールドとして持つ。

actors と messages は、アクターの生成やメッセージの送信を行うために以下のメソッドを持つ。

make アクタープールに新たなアクターを追加するため

```
__device__ void process() {
    int thread_id, block_id;
    Message *msg;
    Actor *act;

    if (block_id == 0)
        参照していないメッセージキューの head を更新
    else {
        if (アクターへのアクセス権を獲得できた場合) {
            switch (メッセージの種類) {
                メッセージにしたがった処理を行う
            }
        } else { // アクターを確保できなかった場合
            メッセージを参照したメッセージキューに送り返す
        }
    }
}
```

図 3 メッセージの処理の疑似コード

の actors のメソッド。

send 新たに送信されたメッセージをメッセージキューに追加するための messages のメソッド。

delete メッセージキューの指定した位置にあるメッセージを消去するための messages のメソッド。

refresh メッセージキューの head を進めるための messages のメソッド。

アクタープールやメッセージキューに格納される、個々のアクターやメッセージも構造体として定義する。アクターの構造体 (以下, actor) は、アクターの ID と状態変数の領域、既知のアクターの ID の集合を含む。メッセージの構造体 (以下 message) には、メッセージの送信先のアドレスと、送信するデータの集合が含まれる。actor と message は、それぞれが持つフィールドのアクセサを持つ。

4. 並列処理の表現

非同期制御を実装するためには、並列技術を組み合わせる必要がある。本節では本研究における実装の上で問題となる部分と、それを解決するための手法を説明する。加えて並行計算部分で処理の高速化させるための方法についても述べる。

4.1 問題と解決手法

非同期制御においてよく問題となるのが **競合状態** である。本システムにおいてアクセス競合の可能性のあるのは、スレッドがメッセージ処理の中で作成したアクターやメッセージを、アクタープールやメッセージキューにコピーをする処理である。どこにコピーをするかは他のスレッドとの兼ね合いを考慮しなければ、競合が発生してしまう可能性がある。

これに対して、競合に対する一般的な手法である排他制御を用いることで解決をする。スレッドはアクタープールまたはメッセージキューの tail の値を排他的に読み取り、

コピーする場所を確保したのち、*tail* の値を更新する。他のスレッドが *tail* の値を読むときには排他的な更新がなされているため、コピー先が重複することはない。CUDA でこの処理を実装するために `atomicAdd` 命令 (A.2.2) を用いた。`atomicAdd` 命令はメモリ上の数値を排他的に決まった数だけ増加させることができる。本システムでは生成したアクターやメッセージの分だけ *tail* の値を増加させる。

アクターの原子性の確保にも排他制御を利用する。各アクターごとにロックを表す大域変数を用意し、開錠状態 (= 0)、施錠状態 (= 1) とする。スレッドがアクターへのアクセスを試みるときにこのロックを見て、開錠状態の場合ロックを排他的に施錠状態へ変更し、処理を続行する。他のスレッドはアクターのロックが施錠状態のときはアクセスできないため、アクターの原子性が確保できる。なおアクターのロックはアクター ID を索引とする配列に保存する。実際の処理には `atomicCAS` 命令 (A.2.2) を用いた。CAS は Compare And Swap の略称であり、すなわちあるメモリ上の数値と指定した値を比較し、等しければメモリの値を別の指定した値に変更、違う場合は何もしないという一連の処理を排他的に行う命令である。ロックの状態に対し、0 と比較することで開錠状態が判定でき、開錠状態の場合はロックの状態を 1 に変更することで施錠状態にする。もし 0 と比較して一致しない場合は施錠状態であるため、何もしない。

ブロック間でのバリア同期を実装することにも排他制御を要する。アクターの原子性は保証されるとはいえ、すべてのスレッドが完全に非同期で動作することは危険をとまなう。本研究では危険性を低減させるためにバリア同期を実装している。バリア同期は `atomicAdd` に加え、`syncthreads` 命令 (A.2.1) を使っている。`syncthreads` はブロック内での同期は保証するが、ブロック間の同期はできない。そこで排他制御と組み合わせることでブロック間同期を実現した。図 4 に擬似コードを示す。バリア同期のための制御変数の一つを用意し、同期ポイントでは各ブロックからの代表スレッドがこの値を 1 ずつ排他的に増加させる。すべてのブロックの代表スレッドが制御変数を更新すると、変数の値は総ブロック数に一致する。制御変数が総ブロック数に一致するまで、先についたブロックをビジーウェイトさせておくことで同期が表現される。なおブロックの代表スレッドのみこの操作を行うわけであるが、ブロック内の他のスレッドが先に進まないようにするために `syncthreads` が利用されている。

コンパイラの最適化による間違ったデータへの参照のような、プログラマから見えない部分で発生する問題もある。CUDA コンパイラは最適化によって、レジスタやキャッシュの利用を優先的に行うのだが、CUDA のメモリモデルではレジスタや L1 キャッシュはブロックごとに独立しているため、他のブロックの処理の結果などを即座に反映で

```
__syncthreads();  
if (threadIdx.x == 0) {  
    制御変数を排他的に 1 増加させる  
    while(制御変数 < 総ブロック数);  
}  
__syncthreads();
```

図 4 バリア同期の擬似コード

きない。競合と同じように正確な値を得ることができない危険性を孕んでいる。

これを解決するために `volatile` (A.1.2) を利用する。`volatile` 宣言された変数に対しては必ず共有メモリまでデータを見に行くため、他のスレッドの計算結果を読み取れる。本研究では、アクターやメッセージのデータ、アクタープールやメッセージキューの管理変数である *head* や *tail* の値は、すべてのスレッドによって変更されうるものであるため、値を読み出すときに `volatile` でキャストしている。

4.2 処理の高速化

処理を高速に行うためには以下のことに注意しなければならない。

- 逐次処理部分の削減
- メモリの読み書きの回数の削減

前節で全スレッドは同種のメッセージを処理するとしたが、これは branch divergence (A.1.1) の回避を目的としている。CUDA の実行モデルは、膨大な数のスレッドを提供するが、実行時はそれらのスレッドを 32 個ずつの *warp* というグループごとにスケジューリングして実行する。*warp* 内のスレッドは同一命令しか実行できず、分岐がある場合は `predicated execution` される。このため分岐はプログラムの並列性を著しく低減する。これを branch divergence と言う。

本システムでは全スレッドが同種のメッセージを処理するため分岐の数が減り、branch divergence を回避している。ただし、空のメッセージをキューより得たスレッドや、メッセージ処理の内に存在する条件分岐は避ける事ができない。branch divergence を回避する技術として、静的に [7]、あるいは動的に [8] コード整理する手法が提案されているが、本システムの実装はコード整理というフェーズを踏むことなく動的に branch divergence を少なくできる。

また排他制御は逐次処理部分を増加させるため、並列性が低下する。そのため排他制御の削減は処理性能を向上させる。例えば本システムにおける *head* の値の更新は排他制御を行わなくともできる。処理されたメッセージの数は簡単に勘定できるので、排他制御を使わずに一回の加算によって *head* の値を更新できる。

なおこれは高速化には関係ないのだが、*head* の値の更新は他の messages の処理がなされているときに行われる。

CPU: Intel Xeon E5645
(6cores/2.4GHz/QPI5.86GT/cache:12MB/TDP:80W)*2
GPU: NVIDIA Tesla C2075 [ETS2075-C6ER] (6GB GDDR5-384bit)

図 5 実行環境

さらに更新は並列性を高めるためにブロックを一つを割いて実行される。このブロックはメッセージ処理に関係なく、*head* の更新のみを行う。ブロック一つに別の処理をさせる発想は、warp 単位で処理の分割を行う研究 [9] から来ている。

メモリの読み書き回数の削減は、現状のシステムでは重要な最適化になる。CUDA でプログラミングするにおいて、処理性能を大きく左右するのはメモリアクセスである。CUDA は複数のメモリ領域を持っているが、メモリによってアクセス速度が大幅に違う (A.1.2)。本研究では、アクターやメッセージをグローバルメモリに置いている。グローバルメモリは容量が大きい反面もっともアクセスに時間がかかる。しかし、アクターやメッセージは大量に使用されるのでこのメモリ領域に置いた。そのためアクターやメッセージの値を読み書きに時間がかかってしまうので、読み書きの回数を削減することは処理時間を減少させることを意味する。アクターやメッセージは固定長であると前節で定義したが、問題サイズによってアクターやメッセージのデータ領域はすべて使われない。使われない領域の初期化も、アクターやメッセージはメモリ上に置かれているためメモリ書き込みである。処理ごとに必要なデータ領域の大きさが分かる場合はそれにしただって初期化を行うことで、メモリアクセスの回数を確実に減らすことができ、処理が高速になる。

5. 評価

この節では二種類の例をもとに、本システムの評価を行う。対象とする例は、フィボナッチ数の計算と、無向グラフの最短距離導出である。

本節を一貫して実行環境は 図 5 である。

5.1 フィボナッチ数の計算

フィボナッチ数の計算はアクターを用いた並行計算の概念の説明にしばしば利用される古典的な例である。フィボナッチ数 (F_n) は再帰的に定義することができるが、各 F_n の計算ごとにアクター (Fib アクター) を用意することで超並列を達成することができる。また、 F_{n-1} と F_{n-2} の計算結果の加算の部分の同期処理においてメッセージ処理の原子性を利用する。この目的で導入する加算専用のアクター (Add アクター) を利用するにあたって、Fib アクターに継続渡しに類似したプログラミングスタイルをする点に特徴がある。以下がアクターを用いた並行フィボナッチの概要である。詳しくは、文献 [1] を参照されたい。

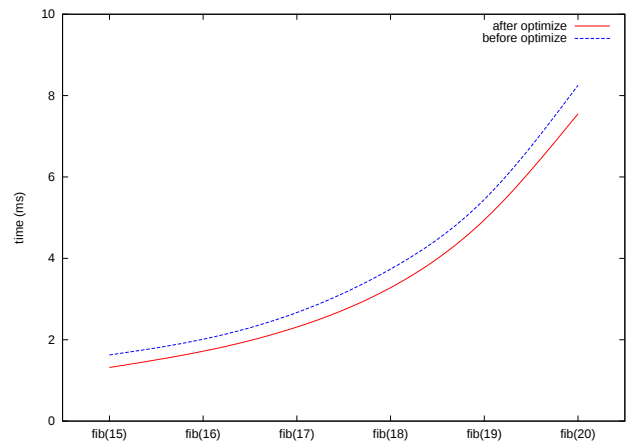


図 6 メモリアクセス回数の改善

Fib アクター $\text{fib}(n, r)$ メッセージを受信する。ここで r はこの計算結果を待つ別のアクターへの参照であり、継続渡し計算流に考えるならば継続を意味する。Fib アクターは $n < 2$ ならば n を r に送信する。 $n \geq 2$ ならば一つの Add アクター ($= a$) と二つの Fib アクターを生成し、それぞれに $\text{fib}(n-1, a)$ メッセージと $\text{fib}(n-2, a)$ メッセージを送信する。これらのメッセージは F_{n-1} と F_{n-2} の計算結果を足し合わせる操作を継続渡しによって表現している。

Add アクター 二つのフィボナッチ数を加算する役割りを果たす。それぞれのフィボナッチ数は $\text{Num}(n)$ メッセージを介して届けられる。Add アクターは受け取った数値を状態変数に記憶し、二つの数値を得た時点で加算を行い、加算結果を継続に辿るアクターに送信する。計算結果を受け取るアクターは Add アクターを作成した Fib アクターが r に初期化しておく。

5.1.1 結果と考察

まず、文献 [1] の並列フィボナッチ計算に忠実に実装した CUDA プログラムによって、 F_{15} から F_{20} を安定的に正しく計算できたことを報告する。ここでのフィボナッチの計算は非同期計算、メッセージ処理の原子性、継続渡しなどアクター計算における中心的な概念を網羅するものであり、この実験が成功したことはアクター計算を GPU 上で正しく実装できることを示すため重要である。

F_{20} の計算過程では、30,000 個以上のアクターが生成され、多くのメッセージを非同期的に交換しながら計算が進捗する。本研究ではアクターのガーベジコレクションを実装していないため、これ以上に規模の大きい計算は控えた。

まず、前節で説明した初期化箇所の低減によるメモリアクセスの回数を減らす最適化の効果を 図 6 に示す。アクタープールとメッセージキューの大きさは 32,768、実行時のスレッドブロックは 5 でブロックあたりのスレッドは 64 である。この結果、0.5 から 1 ms ほど、割合としては

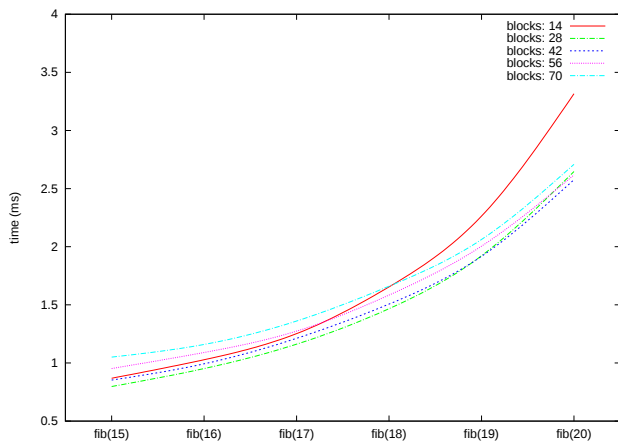


図 7 ブロックの数の変化

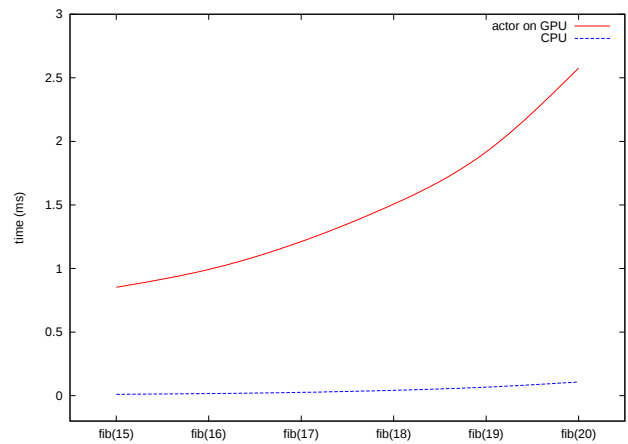


図 9 CPU での処理との比較

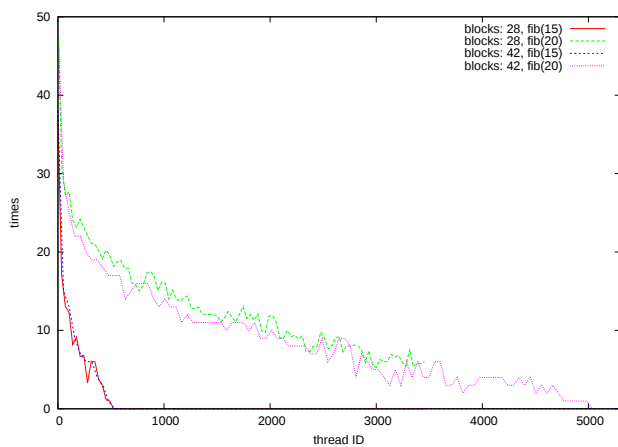


図 8 スレッドごとの処理量

5-10% 程度の効果が見られる。本研究で試みた最適化は、簡単なものが多いが、その性能改善への効果は大きい。

次にブロックの数を変化させることで、どれだけ処理時間に差が出るかを調べた。アクタープールとメッセージキューの大きさは 32,768、スレッドブロックあたりのスレッド数は 128 である。実装は前述のメモリアクセス回数の改善を含んでいる。図 7 を見ると、スレッドブロックの数を増やしても処理があまり速くなっていない。注目すべきはブロック数 28 と 42 のときの処理時間である。F₁₅ では 28 の時の方が速いが、F₂₀ では 42 の方が速い。

この原因を調査するために、ブロック数 28 と 42 の時でスレッドごとのメッセージ獲得成功率を調べた (図 8) と、スレッドごとの処理量がブロックの数にほとんど関わっていない事がわかった。F₁₅ においてはブロック数 28 でも 42 でも、ほぼ同数のスレッドのみがメッセージ処理を実行し、大半のスレッドは動いていない。F₂₀ では、ブロック数 28 の時は全スレッドが処理をしているが、忙しいスレッドと暇なスレッドで偏っている。ブロック数 42 ではやはり同じようにほぼ全スレッドが処理をしており、同様に処理量の偏りも存在する。しかし、ブロック数 42 の方が並列に動作するスレッド数が多いことから F₂₀ の処

理速度が速いと言える。F₁₅ でブロック数 28 の方が処理が速いのは、スレッド数が多い方がバリア同期時の排他制御の部分で時間がかかるからであると考えられる。

すなわち問題サイズが小さいので、あまり差が出なかったと言える。もっと計算を要する問題ならば、処理できるスレッドの数が多いブロック数 42 の方が高速な処理ができるはずである。なお、一度に発行されるメッセージの数が少ないと処理を行えるスレッドの数は限られてしまうため、現在の実装ではロードバランスをとることは難しい。

デバイスメモリ領域量を超えるプログラムは動かすことができないという問題も存在する。実験では F₁₅ から F₂₀ を対象とした。これは先に述べた通り決して大きな計算ではない。本システムではデバイスメモリの量を考慮して、アクタープールやメッセージキューの大きさを制限している。提案したアクターモデルを用いたフィボナッチ数計算プログラムでは、アクタープールが 32,768、メッセージキューが 16,384 で F₂₁ を算出すると、正しい値が返らない。今回のフィボナッチの実装では、アクターを三つずつ生成していくと F₂₁ を計算するには 53,131 のアクタープールが必要になる。アクタープールのサイズを 65,536 すると、正しい値が返る。メッセージキューは使い回しにしているが、仮に処理していないメッセージの数がメッセージキューの大きさを上回ってしまった場合、未処理のメッセージが上書きされてしまい、計算が失敗することもある。

以上よりメモリアクセスの仕方や、メモリ領域の効率の良い使用法を模索する必要があることがわかった。

最後に単体の CPU での計算時間の比較を図 9 に示す。当然ながら CPU の方が高速であるが、メッセージ通信、スケジューリング、同期、分岐 divergence、低いクロック性能があるにも関わらず、並列性が限定的なフィボナッチの計算でここまでの性能を達成できたことは著者らの予想を越えている。

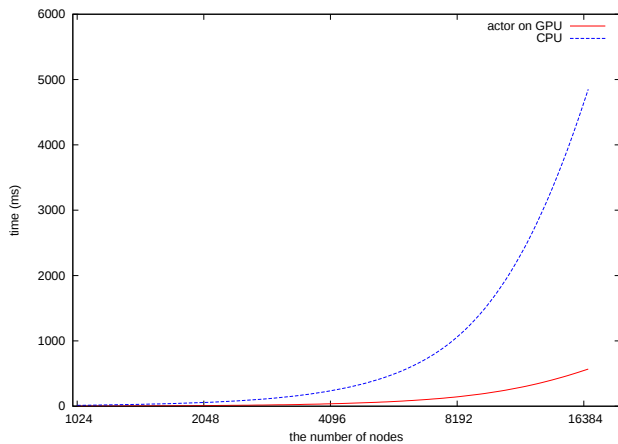


図 10 最短距離導出問題

5.2 無向グラフの最短距離導出

次に行った実験では、Duncan J. Watts の β -モデルによって生成したスモールワールドネットワーク [10] を対象に、一対全の最短経路問題を Dijkstra 法を用いて並列計算した。

β -モデルはクラスタ性が高く、直径の短いネットワークを生成するモデルであり、社会ネットワークの代表的なモデルのひとつである。生成にあたっては、環状に配置したノード群について、まず近隣のノードと相互にエッジで結合する。つぎに、一定の確率 (p) で無作為に抽出したエッジを張りなおすことでスモールワールドネットワークが得られる。

本実験では、スモールワールドネットワークのノードを **Node アクター** として表現し、隣接するアクターへの参照を Node アクターの状態変数に保存することでネットワークを表現した。

Dijkstra 法の実装では、各 Node アクターは起点からの距離を記憶することにする。最初、全 Node アクターは距離を ∞ とする。各アクターはメッセージを介して起点からの最短距離の候補 d を受信し、それがすでに保有する距離より短い場合のみ、自分の距離を d とし、隣接するすべての Node アクターに $d+1$ を送信する。Dijkstra 法の計算は起点に相当する Node アクターに距離 0 を送ることによって開始し、メッセージが枯渇した時点で終了する。

5.2.1 性能評価

比較対象として、CPU で同様の計算を実行した。ノードの数が 1024, 2048, 4096, 8192, 16384 で一対全の最短距離を導出した。フィボナッチ数の計算プログラムで行ったメモリアクセスの減少を適応済みであるスレッドブロックの数はなるべく高速になるものを選び、ブロックが 64 でブロックあたりのスレッドが 64 で実行した。GPU での処理は、ノードの隣接関係を CPU から GPU にコピーし、アクターに情報を与えるところまで含む。

図 10 を見ると、フィボナッチの結果と反対に、アク

ターを利用して実行したものの方が CPU の実行よりもかなり高速であることがわかる。ノード数 16,384 の時には約 8.5 倍の高速化がなされている。このような結果になったのは、ノード数があらかじめ決まっていたため新たなアクター生成が必要でなかったことと、メッセージの種類が一つしかなかったため処理自体が単純であったことが影響していると考えられる。

6. まとめ

本研究では CUDA にアクターモデルを適応させる方法の提案及び実行時システムを実装し、例題を実行することでシステムの評価を行った。フィボナッチの計算では CPU での処理速度に遠く及ばなかったものの、最短距離導出については単体 CPU と比べて 8.5 倍の高速化を実現した。問題のタイプにも依るが、CPU を超える計算ができる点で本システムは有用であると言える。また内容が大きく異なる 2 つの例に対して、メッセージ処理の内容と初期条件、終了条件部分のみ書き換えることで実装できたため、アクターモデルの柔軟性も活かすことができていると判断できる。特にフィボナッチの計算がエラーなく処理できたことは GPU 上でアクターの非同期処理が正しく実装できていることを表し、本研究の未来性を感じることができる。

しかし本稿での実装は未だ稚拙な段階にあり、多くの課題が存在する。現状ではアクター計算が可能になっただけであり、CUDA における様々な最適化手法をほとんど考慮できていない。メモリアクセスを効率的に行うためにグローバルメモリへのアクセスを減らしたり、coalesced access にしたり、シェアードメモリを利用する方法を考えれば処理は高速になる。またメモリ領域の無駄をなくす工夫をし、有限ながらもより多くのアクターやメッセージを扱えるようにすべきである。現在は多くの暇なスレッドも発生してしまっているため、ロードバランスを考慮した実装にする必要がある。

これらの改善でどの程度高速化するかは、本システムに対する多くの有効な実験を行わなければ評価しにくい。本システムの正確な評価を行うことも今後の課題である。またアクターシステムの特性ゆえ、メモリアクセスの改善やロードバランスの確保は実現が非常に難しく、工夫が必要である。場合によっては基礎的なアクター計算の表現の仕方について再考することも考えている。

最後に本研究で未実装の部分を記す。今回の実装では、ガーベジコレクションが未実装である。ガーベジコレクションを実装して、アクタープールを無駄なく利用できるようにする予定である。また実装したのは実行時システムであるため、一般利用することはまだ難しい。メッセージ処理部をシステムに合わせて記述すれば利用できるものの、本システムをよく理解しなければ書けない。アクター言語

のようなものを規定し、本システムを利用した CUDA のコードへの変換を行うコンパイラを構想している。コンパイラは現在作成中である。

謝辞 本研究に際して、お忙しいところお時間を割いていただき、様々なご指導、助言をいただきました額田彰先生、遠藤敏夫先生に深く感謝致します。また、GPGPU についての学びの場を提供していただいている東京工業大学 学術国際情報センターの GPU コンピューティング研究会に感謝致します。

参考文献

- [1] Agha, Gul A.: ACTORS : a model of concurrent computation in distributed systems, The MIT Press, 1986.
- [2] Nvidia Corporation: NVIDIA programming guide, 2012.
- [3] Akinori Yonezawa, Jean-Pierre Briot, Etsuya Shibayama.: Object-oriented concurrent programming ABCL/1, in Proc. OOPSLA '86, pp. 258-268, November 1986.
- [4] Erlang: <http://www.erlang.org/>
- [5] Scala: <http://www.scala-lang.org/>
- [6] 五嶋壮晃, 井出真広, 中田晋平, 倉光君郎: スクリプト言語向け軽量アクターモデルの設計と実装, 情報処理学会研究報告, Vol. 2011-PRO-85, No. 5, pp. 40-40, 2011
- [7] Tianyi David Han, Tarek S. Abdelrahman.: Reducing branch divergence in GPU programs, in Proc. GPGPU-4, March 2011
- [8] 奥山倫弘, 置田真生, 阿部武志, 浅井義之, 野村泰伸, 萩原兼一: インタプリタ型汎用性対シミュレータ insilicoSim の GPU による高速化, 情報処理学会研究報告, Vol.2011-HPC-130, No.9, pp. 1-8, 2011
- [9] Michael Bauer, Henry Cook, Bruce Khailany.: CudaDMA: Optimizing GPU memory bandwidth via warp specialization, in Proc. SC 2011, November 2011
- [10] Duncan J. Watts, Steven H. Strogatz.: Collective dynamics of 'small-world' networks, nature, 1998
- [11] Nvidia Corporation: DYNAMIC PARALLELISM IN CUDA, 2012.
- [12] J. Nickolls, I. Buck, M. Garland, K. Skadron.: Scalable parallel programming with CUDA, in Proc. SIGGRAPH 2008, August 2008

付 録

A.1 NVIDIA GPU と CUDA

NVIDIA GPU のアーキテクチャと、CUDA の基本構造 [2][12] について説明する。

A.1.1 計算モデル

CUDA は NVIDIA によって提供されている GPGPU プログラミング開発環境である。そのため、CUDA は NVIDIA の対応した GPU でのみ動作する。

NVIDIA GPU のアーキテクチャと CUDA のスレッドモデルは密接な関係にある。双方の対応関係について述べる。

GPU は計算単位として複数の *Streaming Multipro-*

cessor (SM) を持つ。SM の中には *Streaming Processor (SP)* というプロセッサが大量にある。

CUDA はある GPU プログラムに対し、膨大な数の **スレッド** で並列に処理を行う。このスレッドの動作を管理するのが、GPU の SP にあたる。またスレッドを管理するために、**グリッド** と **ブロック** という概念を導入する。複数のスレッドを一つのブロックとし、複数のブロックを一つのグリッドとする。ブロックの動作を管理するのが、GPU の SM にあたる。グリッドは GPU のアーキテクチャには対応せず、プログラム一つに対応する構造体と言える。

SP とスレッドは一対一対応せず、SM とブロックも一対一対応しない。SP よりも大量のスレッドが存在し、SM より多くのブロックが存在する。SM 単位でブロックの挙動を制御し、ブロック内のスレッドの動作をスケジューリングして、大量のスレッドを管理する。

スレッドは単独で自由に動作することではなく、*warp* という単位でまとめられて処理を行う。1 *warp* は 32 スレッドである。SM はあるブロックに対し、*warp* 単位でスケジューリングを行う。メモリの使用状況によっては、複数の *warp* を並列に動作させることもできる。

複数のブロックを SM が担当した場合、ブロックあたりのスレッドの数やメモリの使用状況によって、複数のブロックを並列に動作させることができる。ブロック数やブロックあたりのスレッド数、メモリの使用法など上手に考えれば、並列に処理されるブロックの中で *warp* が並列にスケジューリングされて、大きな並列性を生み出すことができる。

プログラマから見て、各スレッドは別個の命令を実行する *SPMD (Single Program Multiple Data)* モデルで動作する。しかし、*warp* は *SIMD (Single instruction Multiple Data)* モデルで処理される。*warp* 内の 32 スレッドは一度に一つの命令のみ実行できる。条件分岐の際は、各スレッドごとの条件にあった処理を並列に処理しているように見えるが、*warp* 内のスレッドが条件分岐ができない。*warp* 内のあるスレッドの処理が終了するまで条件に合わないスレッドが待機する。逆に処理が終了したスレッドは、他のスレッドの処理が終わるまで待機する。そのため *warp* 内のスレッドが条件分岐を起こすと、分岐の数だけ処理に時間がかかる。これを *branch divergence* と呼ぶ。

A.1.2 メモリモデル

CUDA では、スレッド階層ごとに扱うことができるメモリに違いがある。

代表的なものとして、すべてのスレッドから参照可能である **グローバルメモリ**、ブロック内のスレッド間で共有され他のブロックからは見えない **シェアードメモリ**、スレッドごとに独立した **ローカルメモリ** がある。さらにスレッド

ごとに**レジスタ**が割り当てられている。

それぞれのメモリは GPU チップ上に存在するかを含めて、アクセス速度がはるかに違う。メモリ容量にも差がある。グローバルメモリとローカルメモリはチップ外に存在し、データ転送に時間がかかる。その反面容量が大きい。シェアードメモリはチップ上にあるため、アクセス速度は速いものの、容量が小さい。レジスタも当然高速だが、使用できる数は少ない。シェアードメモリやレジスタの使用できる量はブロックごとに決まっている。処理速度を考慮するならばなるべくシェアードメモリを使用し、レジスタの量を超えない実装をする必要がある。

また SM には **L1 キャッシュ**が存在し、すべての SM から参照できるところに **L2 キャッシュ**もある。L1 キャッシュはローカルメモリを、L2 キャッシュはグローバルメモリを参照する際に使用される。キャッシュはコンパイラの最適化によって意図せずに利用され、より高速に処理が行われる。

レジスタやキャッシュは高速な処理を可能とするものの、コンパイラがこれらの使用を優先させるために行う最適化がマルチスレッディングでは問題になる場合がある。いくつかのスレッドで共有するデータが存在すると仮定する。コンパイラは最適化によって、そのデータをレジスタに書き込んでおき、データにアクセスする際にメモリまで参照しない。そのため、あるスレッドがデータに行った変更を他のスレッドが参照できない場合がある。これを避ける方法として、変数を *volatile* 宣言したり、キャストする方法がある。

A.2 CUDA の同期・排他制御

並列処理を行う上でスレッド間でのデータ内容の不一致や、同じメモリ領域への同時書き込みによって正確な値を確保できなくなる場合がある。これを避けるために同期や排他制御などの仕組みが必要不可欠である。CUDA にもそれらを行う機構が存在する。CUDA に組み込まれている同期・排他制御関数 [2] について説明する。

A.2.1 同期制御

スレッドは warp という単位で同一命令を処理することを A.1.1 で説明した。そのため、warp あたりのスレッドは特別に指定せずとも同期を行う。

異なる warp に所属するスレッドは同期を明示的に行う必要がある。CUDA では *syncthreads* という関数によって同期をとることができる。ただしこの関数で同期を行うのは、ブロック内のスレッドに限られる。ブロック内のすべてのスレッドが *syncthreads* を呼ぶまで処理が先に進まなくなる。そのため分岐において、一方の処理列のみ *syncthreads* を含むとすべてのスレッドで関数を呼ぶことが保証されなくなり、正しい処理が行われなくなってしまう。

まう。

ブロック間の同期メカニズムは、CUDA 4 まででは存在していない。同期を取る場合は一度 GPU での処理を停止して、再度 GPU での処理を再開する方法が存在する。

A.2.2 排他制御

排他制御のための関数はいくつか存在するが、ここでは本研究にて使用した関数についてのみ説明をする。これらの関数はデータの読み込み、修正、書き込みを一連の処理として行うことで、排他制御を可能とする。対象となるデータはシェアードメモリかグローバルメモリ上のものである。グローバルメモリ上のデータを処理する場合は、グリッド内のすべてのスレッドに対して排他的に制御ができる。

A.2.2.1 atomicAdd

atomicAdd はある変数のインクリメントを排他的に行う関数である。引数の型によって返り値の型が変化するが、基本形として、引数は排他的に処理を行いたい変数のアドレスと、インクリメント幅の二つを与える。インクリメントされるのは引数として与えたアドレスが指す値であり、関数の返り値としては元の値が得られる。

A.2.2.2 atomicCAS

atomicCAS は "(old == compare) ? val : old" という処理を行う。引数はあるデータ (old) のアドレスと、そのデータと比較するデータ (compare)、一致していた際に old に代入する val という三つを与える。old, compare, val はすべて整数型の変数か値である。関数の返り値は old の元の値である。

A.3 CUDA 5 との関連性

先日 CUDA 5 が発表された。CUDA 5 の新しい機能の 1 つとして、Dynamic Parallelism [11] がある。これは、カーネル関数内で別のカーネル関数を起動するというものである。この機能によって、従来の CUDA では実装できなかった GPU 上での再帰処理も行えるようになった。

本研究で提案したシステムでは、自分自身にメッセージを送ることによって、擬似的な再帰処理を行うことができる。この再帰は再帰呼び出しされる関数がいつ処理されるかわからないという点で擬似的である。