# 分散 GPGPU フレームワーク『ParaRuby』を用いた分散処理実装とその性能評価
# Implementation and Evaluation of Distributed Processings by ParaRuby : a Parallel GPGPU Framework

久原 拓也* 中村 涼† 吉見 真聡 † 三木 光範 † 廣安 知之‡
Takuya Kuhara Ryo Nakamura Masato Yoshimi Mitsunori Miki Tomoyuki Hiroyasu

## 1. はじめに

Graphics Processing Units(GPU) を用いて汎用目的の並列処理を行う手法 (General-purpose computing on graphics processing units; GPGPU) が，近年広く利用されるようになっている [1][2][3]．GPU は CPU と比べて単純ながらも多数のコアを持ち，並列性の高い問題に対して価格あたり，および消費電力あたりの性能が優れている．しかし，GPU を用いてプログラミングを行うには，並列処理やデバイスのメモリ構造などのアーキテクチャに関する専門知識や，各環境独自のプログラミングモデルの学習など，実装に高度な技術を要する．また GPU を導入するにあたり，その初期コストや運用コストが問題になることも少なくないが，最近では，高性能な GPU を搭載したマシンを提供するホスティングサービスが増え始めている．このようなサービスを有効に活用するためにも，ネットワーク上の複数のノードの GPU を利用するための枠組みが求められている．

本論文では，GPGPU プログラミングを支援することを目的として，Ruby を用いた分散 GPGPU フレームワーク ParaRuby を提案する．このフレームワークにより，ネットワークを介した複数ノードの GPU による分散処理が容易に実行可能になる．サーバの GPU 上での処理は Ruby のメソッドとしてクライアント上から呼び出せるようになり，CPU/GPU 間やクライアント/サーバ間のデータ転送がフレームワークにより隠蔽される．

ParaRuby を評価するために，モンテカルロ法による多次元球の求積，およびマンデルブロ集合の計算の 2 つのアプリケーションを用い，ParaRuby の利用の有無で実行時間を比較した．また，サーバを 2 台用いて処理を分割した場合の実行時間を比較し，複数サーバ利用時の処理性能を評価した．

* 同志社大学大学院理工学研究科, Graduate School of Science and Engineering,Doshisha University
† 同志社大学理工学部, Faculty of Science and Engineering,Doshisha University
‡ 同志社大学生命医科学部, Faculty of Life and Medical Sciences,Doshisha University

## 2. GPGPU プログラミング

### 2.1 GPGPU の適用例

GPGPU に関する研究では，GPU の並列処理能力を活かした処理の高速化の事例が多く報告されている．例として，高速フーリエ変換 [4] や，ソーティング [5]，k 近傍探索 [6] などの処理の高速化が研究されている．また，GPU を利用した PC クラスタに関する研究は国内でも積極的に行われている [7]．2011 年 11 月発表のスーパーコンピュータのランキング TOP500 においても，上位 5 台のうち 3 台が GPU を搭載したシステムを採用している [8]．

### 2.2 GPGPU プログラミングの特徴

現在主流となっている GPGPU プログラミング環境としては，GPU ベンダーの NVIDIA が提供する CUDA[9] や，Apple によって提唱された OpenCL[10] がある．両環境とも，C 言語とよく似た構文で記述することができ，GPU のメモリ確保，解放，データ転送，その他各種 GPU の操作に関する機能が提供されている．

GPGPU プログラミングでは，CPU 側の処理と GPU 側の処理の区別を注意しておく必要があり，CPU 側をホスト，GPU 側をデバイスと呼ぶ．ホストによる逐次処理の中でデバイスの並列処理を呼び出し，処理結果を再びホストで利用する形となる．ホストとデバイスの処理は明確に分けて記述し，カーネル関数と呼ばれる特殊な関数に記述された処理がデバイスで処理され，それ以外はホストで処理される．

図 1 に示す通り，GPU の計算資源は grid，block，thread の 3 つの単位で管理される．grid は複数の block をまとめた単位，block は複数の thread をまとめた単位として定義される．1 つの grid は 1 つの GPU に対応する．GPU は複数の Streaming Multi Processor(以下 SM) から構成されており，1 つの SM が 1 つの block に対応する．SM は複数の Streaming Processor(以下 SP) からなり，1 つの SP は 1 つの thread に対応する．

ホストで用意したデータをプログラムの中でデバイスのメモリに転送し，デバイスでの処理結果を再びホストのメモリに転送する必要がある．デバイスで処理を行う際の大まかな処理の流れは以下の通りである．

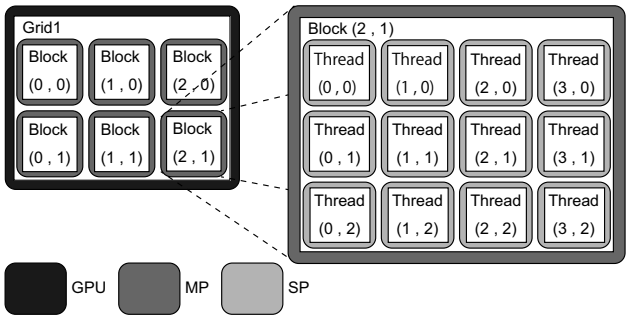


図 1　計算資源の管理方法

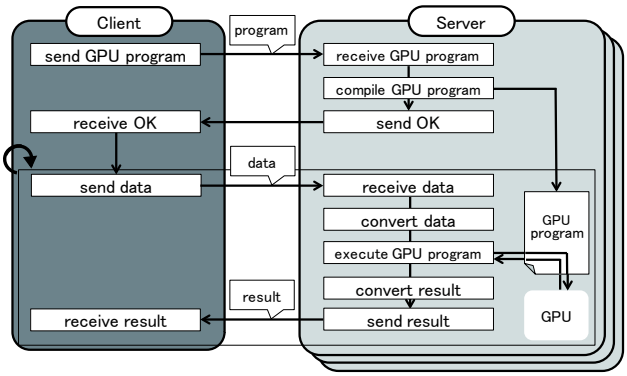


図 2　フレームワークの構造

1. ホストでデバイスのメモリを確保する
2. ホストからデバイスへデータを転送する
3. 転送されたデータをデバイスで処理する
4. デバイスからホストへ処理結果を転送する
5. ホストでデバイスのメモリを解放する

このように，デバイスで処理を行うためにはホストとメモリ間のデータ転送のために多くの処理を行う必要がある．

### 2.3　ビジネス用途への展開

GPU はその高い演算能力から，大きな処理時間を要するアプリケーションにとって魅力的なものとなっており，従来の CPU から汎用計算機能を徐々に奪っていくことが予想される．これはビジネス用途の Web アプリケーション等においてもまた例外ではないと考えられる．

ビジネス用途での GPU 利用のための課題として，特定の目的に用いられる専用のハードウェアを購入するためのコストや先行投資への懸念が考えられるが，そのためのソリューションとして Amazon クラスタ GPU インスタンス [11] などのサービスも登場してきており，高性能な計算資源を搭載したマシンのホスティングが非常に容易になってきている．

## 3.　ParaRuby フレームワークの提案

### 3.1　概要

ParaRuby は，Ruby を利用した分散 GPGPU フレームワークである．GPU を搭載したサーバで予めプログラムを動作させ処理を待ち受けておくことで，クライアントからサーバに対して処理を委譲し，処理結果をクライアントで利用する仕組みを支援する．これにより，GPU を持たないマシンからの GPU 処理や，複数サーバを利用した分散処理が実行可能になる．クライアントとサーバ間の通信は TCP/IP ソケット通信により行われ，デバイスで実行するプログラムとその入出力が転送される．

ParaRuby では，ホストで実行するプログラムは Ruby で記述し，デバイスで実行するプログラム (以下カーネル関数) は OpenCL C 言語または CUDA C 言語で記述する．カーネル関数は，Ruby プログラムの中に文字列として挿入して記述する．

フレームワークの構造を図 2 に示す．ParaRuby ではデバイスとのプロキシとなるクラス (後述の Agent クラス) を提供しており，このクラスのインスタンスをサーバ/クライアント間で共有する．カーネル関数を文字列として共有インスタンスに与えることで，Ruby から呼出可能なメソッドとしてカーネル関数と同名のメソッドが動的に定義される．このインスタンスメソッドをクライアントで呼び出すことで，引数の値を内部的にサーバに転送し，サーバでカーネル関数を実行した結果をメソッドの返り値として返す．これにより，サーバやデバイスとの通信はメソッド呼び出しという形で隠蔽され，カーネル関数の定義部分以外は通常の Ruby プログラムと同様に記述することができる．

### 3.2　設計

ParaRuby は，Ruby のメソッド呼び出しをネットワーク上に拡張するための dRuby と，サーバ側で処理を待ち受けカーネル関数を生成および実行するためのプログラムで構成されている．サーバでは CUDA と OpenCL 用にそれぞれ CudaAgent クラスと OpenCLAgent クラス (以下 Agent クラス) を持っている．

Agent クラスでは，カーネル関数を定義したプログラムをコンパイルするための機能と，任意の名前のカーネル関数を Ruby のメソッドとして実行するための機能を提供している．サーバで提供している Agent クラスの機能を dRuby を介してクライアントから呼び出すことで，ParaRuby の機能は実現されている．フレームワークの機能はサーバにのみ存在しており，クライアントは Ruby に標準添付されている dRuby を用いてサーバの機能を呼び出すだけで良いことから，クライアント側ではフレームワークの利用にあたってライブラリのインストールを行う必要がないという利点がある．

カーネル関数を定義したプログラムをコンパイルする機能は，CUDA の Ruby バインディングである SGC Ruby CUDA と，OpenCL の Ruby バインディングである Barracuda を利用している．両者のインターフェースは統一されており，ク

ライアント側では違いを気にすることなく利用することができる．

任意の名前のカーネル関数を Ruby のメソッドとして実行する機能は，指定したメソッドが見つからなかった場合に必ず呼び出される，Ruby の method_missing という機能を利用して実現している．

また method_missing によりカーネル関数実行前に引数を処理し，各 Ruby バインディングの実行に適した値に変換する処理を行うことで，型の指定などを自動的に行うようにしている．明示的に型が指定されている場合は，カーネル関数の実行前にその型への変換を行う．

### 3.3 コード例

ParaRuby を利用して GPGPU プログラミングを行う例として，単純な積和演算を行うプログラムを図 3 に示す．CUDA および OpenCL のどちらでも記述が可能だが，ここでは OpenCL を用いてカーネル関数を定義する場合の例を示す．カーネル関数 calculate を定義した文字列を Agent クラスの共有インスタンスに渡すことで，calculate という Ruby のインスタンスメソッドがクライアントから利用できる．メソッド calculate に要素数 $n$ の配列を引数として渡すことで，GPU 上で配列の要素数分のスレッドが動作し，calculate で定義した計算が行われる．

```
require 'drb/drb'

# サーバアドレスを指定してインスタンスを共有
agent = DRbObject.new_with_uri(
  'druby://192.168.24.1:3000')

# カーネル関数を定義
agent.program <<EOS
__kernel void calculate(
  __global int *a, __global int *b,
  __global int *c, __global int *d)
{
  // スレッドID を取得
  int id = get_global_id(0);

  // a = b * c + d を実行
  a[id] = b[id] * c[id] + d[id];
}
EOS

# 要素数n の配列を用意
n = 1_000_000
a = Array.new(n)
b = (1..n).to_a
c = (1..n).to_a
d = (1..n).to_a

# カーネル関数で定義した処理を実行
a = agent.calculate(a, b, c, d)
```

図 3　ParaRuby を利用したコード例

## 4. 評価

ParaRuby を用いて GPU で処理を実行したの性能を評価するため，モンテカルロ法によ る多次元球の求積，およびマンデルブロ集合の計算の 2 つのアプリケーションで実行時間 を評価した．評価に用いたマシンの環境を表 1 に示す．マシン間の通信規格には IEEE 802.11n を用いた．

### 4.1 モンテカルロ法による多次元球の求積

モンテカルロ法による多次元球の求積では，$10^5$ 要素の座標点を生成し，異なる次元の球で求積を行った場合の実行時間を測定した．GPU では，各座標点ごとにスレッドを割り当てる．モンテカルロ法による多次元球求積の実行結果を図 4 に示す．

実装方法による実行時間の違いを検証するため，ParaRuby で実装する場合 (ParaRuby)，ParaRuby を利用せず OpenCL で実装する場合 (OpenCL)，C 言語で実装する場合 (C)，および Ruby で実装する場合 (Ruby) の 4 パターンで比較を行った．C と Ruby ではクライアントのみ，OpenCL ではサーバのみ，ParaRuby では両方を利用して実行している．

表 1　実行環境

| | Client | Server |
|---|---|---|
| CPU | Core i7 1.8GHz | Core 2 Duo 2.26GHz |
| GPU | - | GeForce 9400 |
| Memory | 4GB | 4GB |
| OS | Mac OS X 10.7 | Mac OS X 10.7 |

図 4 の縦軸は実行時間，横軸は球の次元数を表しており，ともに対数軸で表している．図 4 を見ると，C 言語は Ruby よりも高速なものの，グラフの傾きはどちらも同程度である．一方，ParaRuby や OpenCL により GPU を利用した実行では傾きが小さく，一定のオーバーヘッドは存在するものの負荷が大きくなるほど GPU の利用が有効であることが分かる．また，ParaRuby と OpenCL の実行時間を比べると，フレームワークのオーバーヘッドによる処理コストが 10〜20% 程度存在することが分かる．

### 4.2 マンデルブロ集合の計算

マンデルブロ集合は，複素平面上の集合が作り出すフラクタルであり，以下の漸化式で定義される複素数列 $\{Z_n\}_{n\in N}$ が n → ∞の極限で無限大に発散しないという条件を満たす複素数 c 全体が作る集合である．$Z$ の各要素はそれぞれ独立して求められることから，並列に計算を行うことができる．GPU の各スレッドごとに $Z_n$ の値を計算することで，求めた値を画素とするようなマンデルブロ集合画像が得られる．

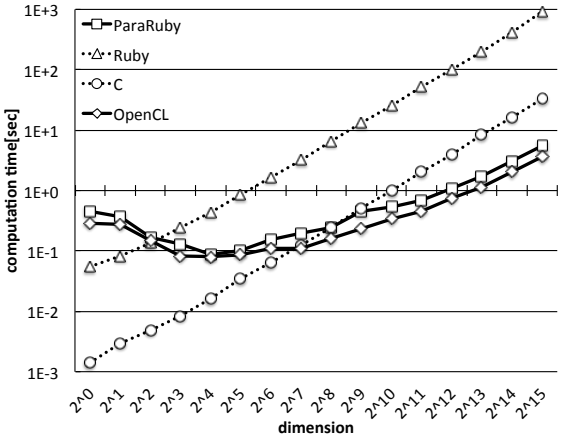


図 4　多次元球求積の実行結果

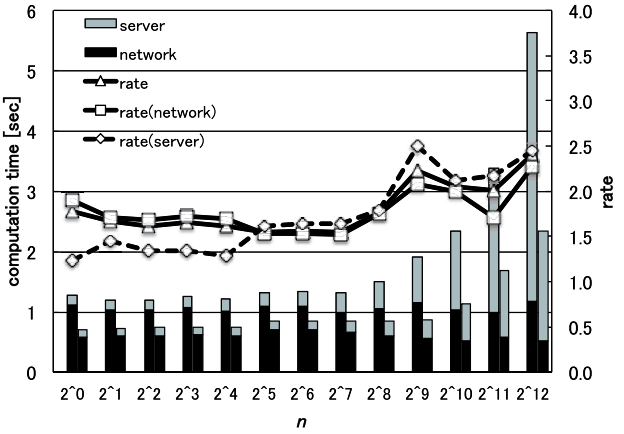


図 5　マンデルブロ集合計算の実行結果

$$\begin{cases} z_{n+1} = z_n^2 + c \\ z_0 = 0 \end{cases}$$

マンデルブロ集合の計算では，$10^5$ 画素に対し，異なる $n$ の値で ParaRuby を用いた場合の計算時間を測定した．またサーバを複数利用した場合の効果を調べるため，2 台のサーバを用いた場合の計測も行った．サーバ 2 台の場合は，処理する画素を分割して各サーバで処理する．

マンデルブロ集合計算に要した時間を図 5 に示す．図中の縦軸は実行時間，横軸は $n$ に与える値を表し，各 $n$ についてサーバ 1 台の場合の結果を左側の棒，サーバ 2 台の場合の結果を右側の棒に表している．network はサーバとクライアント間の通信時間，server はサーバ上での実行時間，rate はサーバ 1 台の場合に比べてサーバ 2 台の場合に何倍の速度向上が見られたかを表している．

通信時間と実行時間の内訳を見ると，通信時間は処理の負荷に関係なく一定であり，実行時間は処理の負荷に応じて変化していることが分かる．

サーバ 1 台の場合とサーバ 2 台の場合を比較すると，サーバの増加により 1.5 倍 2.5 倍程度の速度向上が見られ，負荷が大きくなるほどサーバ増加による速度向上の効果があることが分かる．サーバ 2 台の場合に 2 倍以上の速度向上が見られているが，1 台で処理するデータのサイズが小さくなったことによりメモリキャッシュへのアクセス頻度，およびブロック切替の頻度が減ったことが影響していると考えられる．

## 5. 関連研究

### 5.1 Ruby の GPGPU フレームワーク

Ruby を用いて GPGPU プログラミングを行うことを扱った研究として，Ikra[12] がある．Ikra は，GPU のメモリに実体がある配列を Ruby のクラスとして提供し，配列に対する繰り返し処理などのイテレータを GPU 上で並列実行可能にするものである．Ikra は Ruby を利用した処理系として共通点があるが，Ikra では特定の処理のみが GPU の処理に変換される一方，ParaRuby では記述した任意の処理を実行するという点で異なっている．

### 5.2 GPU での分散処理

複数の GPU ノードを利用した並列実行環境としては，OpenCL を利用したミドルウェアの実装が提案されている [13]．ネットワーク上の複数のノードに搭載された OpenCL アクセラレータを仮想的に 1 つのホストに搭載されているように見せることでノード間通信を隠蔽し，OpenCL のみで複数のノードを利用した分散 GPGPU プログラミング を実現するものである．GPU を搭載した複数ノードを利用できる点で共通点があるが，複数ノードを仮想的に扱える一方，提供されている関数しか扱えない点や，ホスト側も OpenCL に合わせた言語で記述するという点で異なっている．

## 6. まとめ

サーバ上の GPU を利用した GPGPU プログラミングを容易に実現するために，Ruby からサーバ上の GPU で処理を行う仕組みとして，分散 GPGPU フレームワーク ParaRuby を開発した．このフレームワークにより，複数のリモートノードの GPU を利用した並列処理を実現できる．またクライアント自身をサーバとして利用することや，複数のサーバで分散して処理を実行することも可能である．

このフレームワークを用いて 2 つのアプリケーションで評価を行った結果，クライアントの CPU 上のみで処理した場合と比べて，フレームワークからネットワーク上のサーバの GPU を利用した場合に高い実行速度が得られることを確かめた．また，複数のノードに対して処理を分けて実行することでより高い実行速度が得られることを確かめた．

今後の課題として，複数サーバで処理を実行する場合に自動でタスクスケジューリングを行うことや，GPU での計算粒度を変更可能にすること，定型的な処理については Ruby のみで処理を記述できるようにすることなどを検討している．

## 参考文献


[1] 湯川英宜, 平野敏行, 西村康幸, 佐藤文俊. Gpu によるタンパク質高精度静電ポテンシャル計算の高速化. 生産研究, Vol. 61, No. 2, pp. 103–110, 2009.

[2] 成瀬彰, 住元真司, 久門耕一. Gpgpu 上での流体アプリケーションの高速化手法 : 1gpu で姫野ベンチマーク 60gflops 超 (高性能計算とアクセラレータ). 情報処理学会研究報告, 2008-HPC, Vol. 2008, No. 99, pp. 49–54, 2008-10-08.

[3] 東竜一, 藤本典幸, 萩原兼一. Gpu の汎用計算環境 cuda による主記憶上の大規模なテキストに対する高速な全文検索の検討, 2008-hpc. 情報処理学会研究報告, 2008-HPC, No. 19, pp. 139–144, 2008-03-05.

[4] Kenneth Moreland, and Edward Angel. The FFT on a GPU. *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS conference on Graphics hardware*, pp. 112–119, 2003.

[5] Nadathur Satish, Changkyu Kim, Jatin Chhugani, Anthony D. Nguyen, Victor W. Lee, Daehyun Kim, and Pradeep Dubey. FAST Sort on CPUs and GPUs: A Case for Bandwidth Oblivious SIMD Sort. *Proceedings of the 2010 international conference on Management of data*, pp. 351–362, 2010.

[6] Vincent Garcia, and Frank Nielsen. Searching High-Dimensional Neighbours:CPU-Based Tailored Data-Structures Versus GPU-Based Brute-Force Method. *Lecture Notes in Computer Science 5496*, pp. 425–436, 2009.

[7] Tsuyoshi Hamada, Tetsu Narumi, Rio Yokota, Kenji Yasuoka, Keigo Nitadori, and Makoto Taiji. 42 TFlops hierarchical N-body simulations on GPUs with applications in both astrophysics and turbulence. *Conference on High Performance Networking and Computing*, 2009.

[8] TOP500. http://www.top500.org/, 2011.

[9] NVIDIA. Compute Unified Device Architecture Programming Guide, 2007.

[10] John E Stone, David Gohara, and Guochun Shi. OpenCL: A parallel Programming Standard for Heterogenous Computing Systems. *Computing in Science Engineering, 12 Issue:3*, pp. 66–73, 2010.

[11] Amazon Web Services. http://aws.amazon.com, 2011.

[12] 西口裕介, 増原英彦. Gpu 汎用計算を配列イテレータとして記述する ruby 言語処理系の提案. 日本ソフトウェア科学会全国大会第 28 回大会論文集, 1D-2, 2011.

[13] 設樂明宏, 鎌田俊昭, 山田昌弘, 西川由理, 吉見真聡, 天野英晴. 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.