社団法人　電子情報通信学会
THE INSTITUTE OF ELECTRONICS,
INFORMATION AND COMMUNICATION ENGINEERS

信学技報
TECHNICAL REPORT OF IEICE.

PerlのためのCUDAバインディングフレームワークPerCUDA

福本　貴之†　　石浦菜岐佐†

† 関西学院大学 理工学部　〒669–1337 兵庫県三田市学園 2-1

あらまし　本稿では, スクリプト言語 Perl から GPGPU を利用するためのフレームワーク PerCUDA を提案する. PerCUDA では, カーネル関数を Perl で記述し, そこから CUDA の中間言語である PTX を自動生成することによって GPGPU を実現する. PerCUDA によって, Perl プログラミングのみで GPGPU を実現することが可能となる. また, カーネル関数の実行系として GPU と Perl 処理系を切り替えることが可能となり, GPU が搭載されていない環境でも GPGPU プログラムの開発とテストが可能となる. 本手法を実装し, 1024×1024 要素の配列の乗算で実験を行った結果, PerCUDA を利用した GPGPU は, Perl 処理系を利用した場合と比較して 1151.5 倍高速となった.

キーワード　Perl, GPGPU, CUDA

PerCUDA: CUDA Binding Framework for Perl

Takayuki FUKUMOTO† and Nagisa ISHIURA†

† Kwansei Gakuin University,　Gakuen 2–1, Sanda, Hyogo, 669–1337, Japan

Abstract　This article presents "PerCUDA," which is a framework of GPGPU by way of script language Perl. In PerCUDA, kernel functions as well as driver scripts are written in Perl, which are translated into PTX intermediate codes for CUDA and are executed on a GPU upon calls from the drivers. This enables GPGPU programming solely in Perl. An engine for processing the kernel functions can be switched between a GPU and the perl interpreter, which allows developing and debugging scripts on computers without GPUs. A preliminary experiment on array multiplication of 1024×1024 elements demonstrates that GPGPU by PerCUDA is 1151.5 times faster than the Perl interpreter.

Key words　Perl, GPGPU, CUDA

1. はじめに

近年, パーソナルコンピュータやワークステーションの画像処理の高速化を目的に開発された GPU (Graphics Processing Units) を, 数値計算やシミュレーション等の汎用計算に用いる GPGPU (General Purpose computing on Graphics Processing Units) が注目を集めている. GPU は単純な機能を持つコアを多数搭載しており, これらのコアで並列に処理を行うことによって, データ並列性の高い処理を高速かつ低消費電力で実行することができる.

GPU による汎用計算のための開発環境としては, CUDA [1] や OpenCL [2] 等のフレームワークが存在する. しかしこれらのフレームワークでは, 静的型付言語である C 言語を拡張した専用言語でのプログラミングが要求される. 更に, GPU の並列処理能力を活用するためには, 高度なプログラミング知識と専門知識が必要となるため, プログラムの開発コストが高くなるという課題がある.

そこで, Python や Ruby, Perl 等のスクリプト言語から GPU の計算資源を活用する手法が提案されており, 複数の実装が公開されている [3–7]. スクリプト言語は, 文字列の処理やデータの入出力が C 言語に比べて容易であり, 型を動的に決定するため, 型を意識しないプログラミングが可能である. また, コンパイルを必要としないためデバッグも容易である. これらの特長によって, GPU を利用した汎用計算プログラムを低コストで開発できると期待できる.

PyCUDA [3], SGC Ruby CUDA [4], ParaRuby [5] は, GPU 用コードを Python や Ruby のスクリプト中に埋め込むことによって GPGPU を実現する. この手法では, GPU 用コードを直接チューニングできるが, その開発とチューニングには GPGPU プログラミングに関する知識が必要となる. これに対して Ikra [7] は, Ruby で記述されたカーネル関数から GPGPU のためのコードを自動生成する. この手法では, 全てのコードをスクリプト言語で記述できるため, GPGPU を容易に実現できる.

一方, スクリプト言語の一つである Perl を対象とした GPGPU フレームワークとしては, CUDA::Minimal [6] 等が提案されて

いる．CUDA::Minimal は PyCUDA 等と同様に, Perl のコード中に CUDA による GPU 用コードを埋め込むことによって GPGPU を実現するものであり, Ikra のようにカーネル関数を Perl で記述することはできない．また, CUDA::Minimal は Perl で GPGPU プログラミングを行うために必要な最低限の機能しか提供しておらず, Perl のデータ構造とバイナリの変換機能等をユーザ側で実装しなければならない．

そこで本稿では, 全てのコードを Perl で記述できる GPGPU フレームーワーク PerCUDA を提案する．PerCUDA では, Perl で書かれたコードのうち, GPU を利用して並列処理したい部分を, CUDA の中間言語である PTX に変換することによって Perl を利用した GPGPU を実現する．CUDA::Minimal では提供されていなかった, Perl のデータ構造とバイナリ変換機能等のユーティリティーも提供する．また, GPU を利用して並列処理したいコードは GPU だけでなく Perl 処理系でも実行できるので, GPU が搭載されていない環境でもプログラムの開発とテストが可能となる．

提案手法を実装し, 配列の乗算およびマンデルブロ集合の計算を対象に評価を行った．PerCUDA を利用した GPGPU は, 1024 × 1024 要素の配列の乗算では, CUDA を利用した GPGPU に比べて 5.2 倍の計算時間が必要となったが, 全て Perl 処理系で実装した場合と比較すると 1151.5 倍高速となった．またマンデルブロ集合の計算では, Perl 処理系を利用する場合に比べ, 18.3 倍〜2004.4 倍の高速化を実現できた．

2. GPGPU プログラミング

2.1 GPGPU フレームワーク

GPGPU プログラミングのためのフレームワークとしては, GPU ベンダーの NVIDIA が提供する CUDA (Compute Unified Device Architecture) [1] や, Apple が中心となって開発している OpenCL [2] 等が存在する．

いずれのフレームワークでも, CPU (ホスト) 側で行う処理と, GPU (デバイス) 側で行う処理の両方をプログラミングしなければならない．このうち, デバイス側で並列に実行する処理は, カーネル関数と呼ばれる特殊な関数として記述する．カーネル関数以外のコードは全てホスト側で逐次処理されるが, ホスト側のコードでカーネル関数を呼び出すと, GPU はカーネル関数に記述された処理を並列実行し, ホスト側はその計算結果を受け取って逐次処理を再開する．

ホスト側でカーネル関数を呼び出す際には, デバイス側に計算データを格納するために十分なメモリを確保した後に, ホスト側からデバイス側へ計算データを転送しなければならない．またカーネル関数を実行した後は, 逆にデバイス側からホスト側へ計算結果を転送し, 確保したデバイス側のメモリを開放する必要がある．

メモリの確保, 開放, データの転送, カーネル関数の呼び出し等の GPU を操作する機能は GPGPU フレームワークが提供しているので, これらの関数を利用してホスト側のコードを実装する必要がある．

2.2 CUDA を利用した GPGPU

GPU は, 演算資源として複数個の Streaming Multiprocessor (SM) を持ち, 1 つの SM は複数個の Streaming Processor (SP) から構成される．NVIDIA が提供する GPGPU フレームワーク CUDA は, GPU の演算資源を grid (1 つの GPU), block (1 つの SM), thread (1 つの SP) の 3 つの単位で管理する．CUDA のカーネル関数では, grid, block のサイズおよびカーネルが動作している block と thread の ID を, CUDA が提供するビルドイン変数から取得できる．

NVIDIA が提供する GPGPU フレームワーク CUDA は, カーネル関数をコンパイルする nvcc を提供している．nvcc は, GPGPU に特化した C 言語の拡張である CUDA C 言語を, CUDA の中間言語 PTX や, GPU 向けのバイナリに変換することができる．nvcc は, PTX を生成する基盤として, clang [8] のバックエンドとして採用された LLVM (Low Level Virtual Machine) を採用している．そのため, CUDA C 言語だけでなく, C 言語などの LLVM がフロントエンドを持つ言語から, PTX を生成することが可能である．

また CUDA は, デバイス側で並列実行するカーネル関数を呼び出すための API として, ランタイム API とドライバ API と呼ばれる 2 つの API を提供する．ランタイム API を利用する場合, ホスト側のコードを gcc や clang でコンパイルしたものを, デバイス側のコードを nvcc でコンパイルしたものとリンクするか, 両方のコードを nvcc でコンパイルしなければならない．一方, ドライバ API はモジュール機能に対応しており, PTX 等で記述されたカーネル関数を, ホスト側のコードから動的に呼び出すことが可能となる．

3. 関 連 研 究

PyCUDA [3] は Python 向けの GPGPU フレームワークであり, Python のコード中に CUDA C 言語を埋め込むことができる．CUDA C 言語のコードはスクリプト実行時にコンパイルされてカーネル関数となり, Python と連携して GPU を利用した計算を実行する．

SGC Ruby CUDA [4] は Ruby 向けの GPGPU フレームワークであり, PyCUDA と同じく Ruby スクリプトに埋め込んだ CUDA C 言語のコードをカーネル関数に変換して, GPGPU を実現する．

ParaRuby [5] は Ruby 向けのフレームワークだが, ネットワーク上の複数ノードの GPU 資源を利用できる分散型の GPGPU フレームワークという特長を持つ．そのため ParaRuby では, クライアントに GPU が存在しない場合でも, サーバ上の GPU でカーネル関数を実行することが可能である．

これらのフレームワークは, フレームワークが対象とする言語で記述されたスクリプトの中に, カーネル関数のコードを埋め込むことにより, GPGPU を実現している．この方式は, GPGPU に特化した言語でカーネル関数を記述するのでチューニングは容易だが, カーネル関数の記述とチューニングには, GPGPU に対する専門知識が必要となる．また, カーネル関数の粒度を変更する場合, フレームワークが対象とする言語とカーネル関

数を記述する言語の間でコードのマイグレーションを行わなければならない.

これに対し, Ruby を対象とする GPGPU フレームワークである Ikra [7] は, Ruby のコードから自動的に CUDA C 言語によるカーネル関数を生成することにより, GPGPU を実現する. この方式では, カーネル関数の細かなチューニングが難しいものの, GPGPU を利用したプログラムを容易に実現することができる.

一方, Perl 向けの GPGPU フレームワークとしては, Perl 用のモジュールとして提供されている CUDA::Minimal [6] 等がある. CUDA::Minimal は PyCUDA のように, Perl スクリプト中に CUDA のカーネル関数を埋め込み, CUDA のランタイム API を利用して GPGPU を実現する Perl モジュールである. しかし, CUDA::Minimal は, Perl スクリプトに記述したカーネル関数の起動と, そのために必要なメモリの確保, 開放, ホストとデバイス間のデータ転送を行う機能しか提供していない. そのため, ホストからデバイスへデータを転送する際に必要となる Perl のデータ構造のバイナリ化や, デバイスから取得した計算結果のバイナリを Perl のデータ構造に変換する機能は, ユーザが記述しなければならない.

4. カーネル関数の自動生成による GPGPU

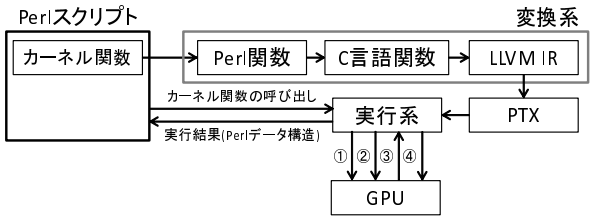
4.1 概　　要

本稿では, CUDA を利用した Perl 向け GPGPU フレームワーク PerCUDA を提案する. PerCUDA は, Perl で記述したカーネル関数から PTX を自動生成することにより GPGPU を実現する. そのため, Ikra と同様, ホストで処理するコードとデバイスで処理するコードの両方を Perl で記述することができる. また, GPU が搭載されていない環境では, カーネル関数の処理系として Perl 処理系を利用することによってスクリプトを実行可能にする.

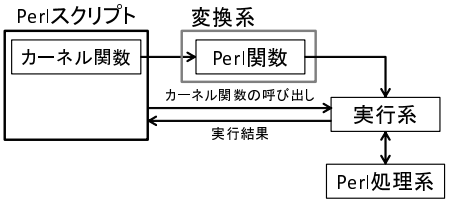
PerCUDA フレームワークは, Perl で記述したカーネル関数を変換する変換系と, 変換系が生成したコードを実行する実行系から成る. PerCUDA の処理の流れを 図 1 に示す. (a) はカーネル関数の処理系として GPU を利用する場合, (b) は Perl 処理系を利用する場合である.

カーネル関数の処理系として GPU を利用する場合 (a), 変換系は Perl で記述したカーネル関数を一旦独立した Perl の関数として切り出す. この Perl の関数を C 言語に変換し, そこから LLVM フレームワークの中間表現 LLVM IR を経由して, カーネル関数の PTX を生成する. Perl スクリプトからカーネル関数が呼び出されると, 実行系はデバイスである GPU に対して, 以下の処理を行う.

（1） 引数として与えられた Perl のデータ構造のバイナリ化
（2） デバイス側のメモリ確保
（3） ホスト側からデバイス側への計算データの転送
（4） カーネル関数のロードと実行
（5） デバイス側からホスト側への実行結果の転送
（6） デバイス側のメモリ解放



(a) GPU 利用時



(b) Perl 処理系利用時

図 1　PerCUDA の処理の流れ

（7） 実行結果のバイナリデータを Perl のデータ構造に変換

カーネル関数を Perl 処理系で実行する場合 (b) は, 変換系が生成する Perl 関数をそのまま Perl 処理系で実行する.

4.2 PerCUDA の記法

PerCUDA フレームワークを利用したプログラムの例 (配列の乗算の計算) を図 2 に示す.

カーネル関数のコード等は, 9〜31 行目のように, PerCUDA モジュールの new メソッドで定義する. new メソッドでは, Perl で記述したカーネル関数 (22〜28 行目) の他, GPU で処理するデータの次元 (10 行目), サイズ (11 行目), 引数とそのデータ型 (13 行目〜18 行目), 返り値 (19 行目) を指定する. カーネル関数を実行する処理系は, 12 行目のように engine として指定する. engine の値が「gpu」の場合は GPU で, 「perl」の場合は Perl 処理系で実行することを意味する. 処理系の指定がない場合, PerCUDA は GPU の搭載を自動的に検出し, GPU が搭載されていれば GPU を, そうでなければ Perl 処理系を利用する.

new メソッドで定義したカーネル関数は, run メソッド (33 行目) で実行する. run メソッドは, 指定された処理系でカーネル関数を実行し, new メソッドで指定した返り値を返す.

カーネル関数の内部で利用可能な Perl のデータ構造はスカラー変数と 3 次元までの配列であり, ハッシュは利用できない. ただし新しい配列を宣言することはできず, カーネル関数に対して引数として渡された配列のみ使用できる. 一方, スカラー変数については, カーネル関数の内部で新たに宣言できる.

演算に関しては, 四則演算 (加算, 減算, 乗算, 除算, 剰余残), インクリメント, デクリメント, ビット演算, 論理演算, 数値の比較演算を利用することができる. 制御構造は, 条件分岐 (if, elsif, else) および繰り返し (for, while) に対応している. Perl には, 制御構造に関するコマンドとして繰り返し処理を中断する last コマンド, 現在の処理を中断して次の処理を開始する next コマンドが提供されているが, PerCUDA では last コマンドのみ対応している.

```
01: use PerCUDA;
02:
03: my $array = [
04:   [ 1 .. 3 ],
05:   [ 4 .. 6 ],
06:   [ 7 .. 9 ],
07: ];
08:
09: my $runner = PerCUDA->new(
10:   dim => 2,          # 配列の次元
11:   size => 'array1', # 配列のサイズ
12:   engine => 'gpu',  # 使用する処理系
13:   argument => [     # 引数とデータ型
14:     array1 => 'f',
15:     array2 => 'f',
16:     array3 => 'f',
17:     N      => 'i',
18:   ],
19:   retval => ['array3'], # 返り値
20:   source => <<'KERNEL', # カーネル関数
21:
22: my $tmp = 0;
23: for (my $i = 0; $i < $N; $i++) {
24:   $tmp = $tmp
25:     + $array1->[$i]->[$y]
26:     * $array2->[$x]->[$i];
27: }
28: $array3->[$x]->[$y] = $tmp;
29:
20: KERNEL
31: );
32:
33: my $retval = $runner->run($array, $array, []);
```

図 2　PerCUDA を利用した GPGPU プログラミングの例

カーネル関数では文字列をそのまま処理できないため, 文字列リテラルを利用することはできない. また, 文字列に関連する演算子や組み込み関数も利用できない.

4.3 変 換 系

変換系は, Perl で記述したカーネル関数のコードを PTX に変換する. 図 2 の記述に対するコード変換の例を図 3 に示す. Perl で記述した カーネル関数 (a) は, 変換系によって Perl 関数 (b), C 言語関数 (c) に変換する. (c) のコードは, clang を活用して LLVM の中間表現 LLVM IR に変換し, LLVM が提供する llc を利用して PTX を生成する.

(b) のコードでは, new メソッドで指定した引数の他に, `$size_x` と`$size_y` という変数が引数として加わっているが, この変数には new メソッドで指定した配列のサイズが格納される. (c) では, カーネル関数が動作している thread の ID を取得しなければならないが, C 言語からは CUDA が提供するビルドイン変数が利用できない. そこで, blockIdx, blockDim, threadIdx をプレフィックスとして持つ, 実体のないテンポラリ関数を利用して thread の ID を生成する. これらの関数は, LLVM IR の段階で, それぞれ CUDA のビルドイン関数を参照するように書き換える.

なお, 実行時にコード生成を行うと時間がかかるが, これは変換系が生成したコードをキャッシュすることにより解決する. 変換前の Perl コードが変化していなければ, キャッシュしたコードを再利用し, 2 回目以降の実行時間の削減を図ることができる.

4.4 実 行 系

実行系は, ホスト側のコードが run メソッドを呼び出した際に起動される. カーネル関数の実行系として, GPU を利用する

```
1: my $tmp = 0;
2: for (my $i = 0; $i < $N; $i++) {
3:   $tmp = $tmp
4:     + $array1->[$i]->[$y]
5:     * $array2->[$x]->[$i];
6: }
7: $array3->[$x]->[$y] = $tmp;
```

(a) カーネル関数の記述

```
01: sub kernel_function {
02:   my ($array1, $array2, $array3,
03:       $N, $size_x, $size_y) = @_;
04:   for my $x (0..$size_x - 1) {
05:     for my $y (0..$size_y - 1) {
06:       my $tmp = 0;
07:       for (my $i = 0; $i < $N; $i++) {
08:         $tmp = $tmp
09:           + $array1->[$i]->[$y]
10:           * $array2->[$x]->[$i];
11:       }
12:       $array3->[$x]->[$y] = $tmp;
13:     }
14:   }
15: }
```

(b) Perl 関数化されたカーネル関数

```
01: void kernel_function
02:   (float* array1, float* array2, float* array3,
03:    int N, int size_x, int size_y)
04: {
05:   int x =
06:     blockIdx_x() * blockDim_x() + threadIdx_x();
07:   int y =
08:     blockIdx_y() * blockDim_y() + threadIdx_y();
09:   float $tmp = 0;
10:   for (int i = 0; i < N; i++) {
11:     tmp = tmp
12:       + array1[i + y * sise_x]
13:       * array2[x + i * size_x];
14:   }
15:   array3->[x + y * size_x] = tmp;
16: }
```

(c) C 言語に変換されたカーネル関数

図 3　変換系によるコードの変換

場合の処理の流れは, 次の通りである.

（1） 引数のバイナリ化
ホスト側からデバイス側へのデータ転送は, バイナリ形式で行わなければならないため, 引数として与えられた Perl データは, 指定された型のバイナリへ変換する.

（2） デバイス側のメモリ確保
生成したバイナリから必要となるメモリ領域を計算し, デバイス側へメモリを確保する.

（3） ホスト側からデバイス側への引数の転送
デバイス側で確保したメモリ領域に対して, バイナリに変換した計算データを転送する.

（4） カーネル関数のロードと実行
変換系が生成した PTX をロードし, API を利用してカーネル関数を実行する.

（5） デバイス側からホスト側への実行結果の転送
デバイス側からホスト側へ, バイナリ形式の実行結果を転送する.

（6） デバイス側のメモリ解放
デバイス側に確保したメモリ領域を開放する.

（7） 実行結果の Perl 構造化

バイナリ形式の実行結果を, Perl のデータ構造に変換した上で返り値とする.

なお, カーネル関数の処理系として Perl 処理系を利用する場合には, run メソッドに与えられた引数をそのままカーネル関数から生成された Perl の関数へ渡し, 関数が返した値を返り値として利用する.

5. 実装と評価

5.1 実　　装

提案手法を Perl と CUDA フレームワークを利用して実装した. なお, PerCUDA が対応する CUDA のバージョンは 5.0 である. Perl から CUDA のドライバ API を操作する機能は, Perl と C 言語のコード間拡張インターフェイスを作るための機構 XS を利用し CUDA::DriverAPI というモジュールとして実装した. CUDA::DriverAPI は, デバイス側のメモリの確保, 開放, PTX のロード, カーネル関数の実行等の機能を, Perl の関数として提供する. カーネル関数を実行する処理系として GPU を利用する場合, CUDA::DriverAPI から CUDA のドライバ API を利用して, GPU の操作を行う.

5.2 実 験 結 果

PerCUDA の性能を評価するため, 配列の乗算とマンデルブロ集合の計算の実行時間を計測した. 実験は Linux (Ubuntu), Intel Core i7 3.90GHz, メモリ 16.0GB, NVIDIA GeForce GTX 660Ti, Perl 5.16.3 の環境で行った.

表 1 は, 1024 × 1024 の要素を持つ配列の乗算の実行時間である. 「GPU」は PerCUDA においてカーネル関数の処理系として GPU を利用した場合, 「Perl 処理系」はカーネル関数の処理系として Perl 処理系を利用した場合, 「CUDA」は PerCUDA を使わず, 全てのコードを CUDA C 言語で実装した場合の実行時間である. また,「前処理」はカーネル関数に対する引数の確認および Perl のデータ構造のバイナリ化および PTX のロードにかかった時間を示す. 「転送」はホスト側からデバイス側およびデバイス側からホスト側へのデータ転送にかかった時間, 「実行」はカーネル関数の実行にかかった時間,「後処理」は GPU の実行結果であるバイナリを Perl のデータ構造に変換するためにかかった時間を示す.

表 1　配列の乗算 (1024 × 1024) の実行時間

	PerCUDA		
	Perl	GPU	CUDA
前処理	0.0000401s	0.0435131s	-
転送	-	0.0218434s	0.0029387s
実行	232.7142642s	0.0333679s	0.0358331s
後処理	-	0.1033700s	-
合計	232.7143043s	0.2020944s	0.0387718s

実行時間は, PerCUDA で実装した GPGPU の場合, CUDA C 言語で実装した場合と比べると 5.2 倍の時間が必要となった. これは, PerCUDA の前処理および後処理に費やす時間が原因である. 前処理では Perl のデータ構造をデバイスへ転送するためにバイナリ化する処理, 後処理ではデバイスから転送したバイナリを Perl のデータ構造に戻す処理があり, これらの時間が負担となっている. しかし, PerCUDA で実装した GPGPU は, Perl 処理系を利用した場合と比べると, 1151.5 倍高速となっている.

表 2 は, 10^6 画素に対してマンデルブロ集合の計算を行った際の計算時間である. マンデルブロ集合は, 以下の漸化式で定義される複素数列が, $n \to \infty$ の極限で無限大に発散しない条件を満たす複素数である c 全体が作る集合である.

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

「GPU」は PerCUDA の処理系として GPU を利用した場合,「Perl」は Perl 処理系を利用した場合である. $n = 2^0$ の場合, GPU と Perl の性能差は 18.3 倍だが, $n = 2^{12}$ の場合, GPU は Perl 処理系を利用した場合と比べて 2004.4 倍の高速化となっている.

表 2　マンデルブロ集合の計算の実行時間

	PerCUDA	
n	GPU	Perl
2^0	0.14419s	2.63360s
2^1	0.14523s	3.27114s
2^2	0.14588s	4.24830s
2^3	0.14409s	5.49464s
2^4	0.14605s	7.26325s
2^5	0.15051s	10.08421s
2^6	0.15296s	15.55460s
2^7	0.15346s	26.60217s
2^8	0.15998s	49.52719s
2^9	0.17723s	91.99616s
2^{10}	0.20646s	175.36165s
2^{11}	0.26290s	348.06578s
2^{12}	0.34298s	687.46580s

6. む　す　び

本稿では, CUDA を利用した Perl 向け GPGPU フレームワーク PerCUDA を提案した. 本手法を実装し, Perl で記述したカーネル関数から自動生成したに PTX を, CUDA のドライバ API を利用して実行できることを確認した. また, 1024 × 1024 の要素を持つ配列の乗算で実験を行った結果, PerCUDA を利用した GPGPU は, Perl 処理系を利用した場合と比較すると 1151.5 倍高速となった.

今後の課題としては, カーネル関数の処理系として GPU を利用した場合の前処理と後処理の高速化, カーネル関数に記述できる Perl の構文の追加, Perl から C 言語への変換時における型推定の精度向上等が挙げられる.

なお本研究の成果物である PerCUDA および CUDA::DriverAPI は, Perl のモジュールアーカイブである CPAN [9] および GitHub で公開する予定である.

謝辞　本研究を進めるにあたり GPGPU と CUDA について御助言を頂きました 首都大学東京 川上大喜氏, 実装にあたってご助言を頂きました 関西学院大学 中村遼氏, および関西学院大学 石浦研究室の諸氏に感謝します.

文　献

[1] NVIDIA: Computing Unified Device Architecture Programming Guide (2007).
[2] J. E. Stone, D. Gohard, and G. Shi: “OpenCL: A Parallel Programming Standard for Heterogenous Computing Systems,” in *Conputing in Science Engineering*, Vol. 12, No. 3, pp. 66–73 (2010).
[3] A. Klöckner, N. Pinto, Y. Lee, B. Catanzaro, P. Ivanov, and A. Fasih: “PyCUDA: GPU Run-Time Code Generation for High-Performance Computing,” in *Parallel Computing*, Vol. 38, Issue 3, pp. 157 – 174 (Mar. 2012).
[4] http://rubygems.org/gems/sgc-ruby-cuda/.
[5] 中村涼, 吉見真聡, 三木光範: “Ruby を用いた分散 GPGPU フレームワーク『ParaRuby』の開発と評価,” 信学技報, CPSY2011-76 (Jan. 2012).
[6] https://github.com/run4flat/perl-CUDA-Minimal/.
[7] Hidehiko Masuhara and Yusuke Nishiguchi: “A Data-Parallel Extension to Ruby for GPGPU,” in *Proc. 9th Workshop on Reflection, AOP and Meta-Data for Software Evolution (RAM-SE)*, pp. 3–6 (June 2012).
[8] http://clang.llvm.org/.
[9] http://www.cpan.org/.