日本ソフトウェア科学会第 28 回大会 (2011 年度) 講演論文集

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

西口 裕介　増原 英彦

本研究は GPU 汎用計算記述のための Ruby 言語処理系 Ikra を提案する．Ikra は GPU のメモリに実体がある配列を Ruby の 1 クラスとして提供し，配列に対する map や inject のなどのイテレータを GPU 上で並列実行する．処理系はイテレータに対するブロック引数の Ruby コードを CUDA のカーネル関数へとあらかじめ変換しておき，イテレータ呼出時に CPU/GPU 間のデータ転送を行った後に CUDA コードを実行する．簡単な計算プログラムのプロトタイプ処理系による計算部分 (カーネル) の実行速度は，同等の CUDA プログラムの 0.9〜1 倍，Ruby インタプリタによる実行の 2000〜8000 倍であった．

1　はじめに

GPU(Graphics Processing Units) を用いて数値計算や記号処理などを並列に行う手法 (general purpose computing on graphics processing units: GPGPU) が注目を集めている．これは GPU が CPU と比べて単純ながらも多数のコアを持つため，データ並列性のある問題に対して価格あたり，また消費電力あたりの性能が良くなる場合があるためである．

現在主流の GPGPU プログラム開発方法は C を拡張した CUDA 言語 [6] や OpenCL フレームワーク [5] を用いるものであるが、データの入出力処理の記述や最適化のための試行錯誤などを含めて考えると開発者の負担は大きい．

そこで，本稿は Ruby 言語を用いて GPGPU プログラムを記述する処理系 Ikra[†1] を提案する．Ruby を採用した理由は，文字列や入出力処理などのクラスライブラリが充実しており，自動ゴミ集め機能を備えているためである．これに加えて，GPU による並列計算にも標準 Ruby に近い記法を提案することで GPGPU プログラムの開発を容易ならしめることを目指す．

本稿の構成を以下に示す．2 節では現在の GPGPU プログラミング手法を概観し，その課題を指摘する．3 節では Ikra による GPGPU プログラミング手法を示す．4 節では配列クラスの API 設計を示す．5 節では処理系の実現方式を示す．6 節では小規模なプログラムを用いた Ikra の性能評価を示す．7 節では関連研究について述べ，8 節はまとめの節とする．

2　GPGPU プログラミングの課題

現在 GPGPU プログラムを作成するための言語処理系として表 1 に示すようなものが提案されているが，いずれの処理系でも (1)GPU によって並列に行わせる計算 (以降カーネルと呼ぶ) を特定すること (2)CPU/GPU 間のデータ転送の 2 つが必要となる．以下ではまず GPGPU プログラミングの特徴を述べ，プログラミング上の課題を整理する．

2.1　GPGPU プログラミングの特徴

GPU は多数のコアで同じ処理を並列に行う SIMD 型処理を行う．そのため GPGPU プログラムは CPU による逐次処理と GPU による多数のデータ (以降並

General-purpose computing on GPU in Ruby with array iterators

Yusuke Nishiguchi, Hidehiko Masuhara, 東京大学大学院総合文化研究科, Graduate School of Arts and Sciences, University of Tokyo.

†1　イクラは「海のルビー」ともよばれる．

表 1 GPGPU 言語処理系の比較

	CUDA	hiCUDA	OMPCUDA	JCUDA	Accelerate	PyCUDA	SGC Ruby CUDA	Ikra
逐次実行部の記述言語	C	C	C	Java	Haskell	Python	Ruby	Ruby
カーネルの特定方法	独立関数	プラグマ	プラグマ	CUDA 関数	高階関数	CUDA 関数	CUDA 関数	イテレータ
データ転送 (並列データ)	手動	手動	自動	手動	手動	手動	手動	自動
データ転送 (スカラー値)	手動	自動	自動	手動	自動	手動	手動	自動

列データと呼ぶ) に対する並列処理を交互に行う形となる．カーネルは CPU とは異なる GPU 用機械語へとコンパイルされる[†2]必要がある．プログラムのどの部分をカーネルとするかは自明でなく，問題の大きさ，GPU のコア数，必要とするメモリ量等から最適な部分を選択しなければならない．例えば行列積を計算する 3 重ループを GPU で計算する場合，行列の要素数が GPU で作成できるスレッド数より小さければ，外側の 2 重ループを並列化し，最内ループをカーネルとする (GPU コアに最内ループを逐次実行させる) ことが多い．

また，CPU と GPU はメモリチップを共有しないため，CPU が用意したデータを使って GPU に並列に計算させる，またその逆の場合にはデータを転送をしなければならない．並列データだけでなく，カーネルが計算に用いるパラメータも明示的に渡される必要がある．データ転送のタイミングも自明ではなく，プログラマが最適となるものを決定しなければならない．例えば，ファイルから読み込んだデータを GPU で複数回処理し，結果をファイルに出力するような場合を考える．ファイルから読み込まれたデータはまず CPU メモリに格納されるので，GPU で処理する前に GPU メモリに転送しなければならないし，結果をファイルに出力する前には GPU メモリから CPU メモリへ転送しなければならない．しかし，連続する GPU 処理の間では転送の必要はない．

2.2 GPGPU 言語処理系の課題

表 1 に示した言語処理系は GPGPU プログラミングの負担を低減する目的を持っている．以下では，これらの処理系がとっている手法の概略と残されている課題を述べていく．

†2 例えば，nVidia 社の GPU に対しては CUDA コンパイラ (nvcc) を利用するのが一般的である．

2.2.1 カーネルの特定方法と記述言語

JCUDA [11]，PyCUDA [4]，SGC Ruby CUDA [9] はそれぞれ Java，Python，Ruby のプログラムから CUDA で記述されたカーネル関数を呼び出せるようにした処理系である．この手法ではカーネルは逐次実行部とは別の言語で記述された関数として自明に特定できる．しかしながら，カーネルの粒度を変更する場合などでは，異なる言語間でプログラムを変換しなければならず，プログラマの負担は大きい．

hiCUDA [2]，OMPCUDA [7]，Accelerate [1] はカーネルと逐次実行部を同じ言語 (それぞれ C，C，Haskell) で記述させる．hiCUDA，OMPCUDA ではプログラマが for 文の前にプラグマを挿入すると，処理系はループ内の処理をカーネルとして並列実行する．Accelerate には並列に計算を行う map や fold などの高階関数が用意されており，それらに渡される関数引数としてカーネルが特定される．これらの処理系の GPGPU プログラムは挿入されたプラグマやデータ型の違いを無視すると通常の逐次プログラムと変わらない．逆に言えば，逐次プログラムに対する簡単な挿入や変換で GPGPU プログラムを作成することができる．

2.2.2 データとパラメータの受け渡し

CUDA，JCUDA，Accelerate，PyCUDA，SGC Ruby CUDA は，CPU データと GPU データを別の変数に格納し，その間の転送をプログラマが明示的に行う[†3]必要がある．hiCUDA は 1 つの変数で CPU データと GPU データの両方を表しているが，データ転送はプログラマが明示的にプラグマを挿入して指示する．OMPCUDA は同一の変数にデータを格納し，転送も自動的に行う．転送を自動的に行う OMPCUDA は，プログラムの記述が簡潔になるものの，転送の必

†3 CUDA では cudaMemcpy を呼び出す．

要性を判断するために変換時に多少の時間コストを要する．

また，カーネル内で使用するスカラー値のパラメータに関しても，転送に手動/自動の別がある．CUDA，JCUDA，Accelerate，PyCUDA，SGC Ruby CUDA では，CUDA カーネル関数の引数として明示的に渡さなければならない．一方で，hiCUDA，OMPCUDA，Accelerate では，カーネルへのパラメータはカーネルの外側で定義された変数 (をカーネル内で参照すること) として現れる．これらは処理系がカーネル内の自由変数を自動的に発見して受け渡す．

3 提案

表 1 の右端の列に示すように，提案する Ikra は，Ruby 言語に基づいた GPGPU 言語処理系である．逐次実行部，カーネル部の両方を Ruby で記述させることで，Ruby 言語に備わる入出力処理，記号処理などのための豊富なライブラリを使いつつ，GPGPU プログラムを容易にプロトタイプできる処理系を目指す．Ikra は並列データを表すクラスを用意し，そのクラスのオブジェクトに対するイテレータ呼出 (map や inject など) に渡されるブロック引数をカーネルとして GPU 上で並列に実行する．Ruby ではもともと for のような繰り返し構文よりも配列オブジェクトに対するイテレータを用いることが一般的であるため，カーネルの自然な記述が可能となる．並列データの CPU/GPU 間転送は，実行時の最初のアクセス時に自動的に行い，また Ruby のゴミ集めと連動して GPU メモリの解放をすることでプログラマの負担を軽減する．カーネル中のパラメータはイテレータブロック内の自由変数としてあらわれるが，これらもイテレータ呼出時に自動的に渡される．

Ikra が GPU で実行するプログラムの例としてマンデルブロ集合の計算を図 1 に示す．標準 Ruby と異なるのは，配列クラスとして Array の代わりに Ikra が独自に提供する MDarray を用いている点である．

MDarray オブジェクトは GPU メモリ上のデータを表す．これに対するイテレータ呼出があると，GPU の各コアが各要素に対してブロック内を並列に実行する．この例は GPU の各コアがコンストラクタ new に与えられたイテレータブロックを，2 次元の添え字を引数として並列に実行する．結果の配列は GPU メモリ上にできるが，この後に画面やファイルに出力するコードが実行されると CPU メモリに転送されてから，CPU 上の Ruby インタプリタによって処理される．

4 設計

MDarray は整数または実数のどちらかを要素型とする多次元配列を表現するクラスである．各メソッドの機能を以下に述べる．

生成

MDarray の生成は要素値を GPU で計算する方法と CPU で生成したものを転送する方法があり，MDarray.new メソッドに与える引数によって使い分ける．

- MDarray.new($x_1, x_2, ..., x_n$){ $|i_1, i_2, ..., i_n|$ e }

```
require "MDarray"

#res, r_min, r_max, i_min, i_max,
#limit, inf, r_size, i_sizeを設定

result = MDarray.new(r_size, i_size){|i,j|
  #計算する点の座標
  cr = r_min + res*i; ci = i_min + res*j

  iter = 0; zr = 0.0; zi = 0.0

  while(iter<limit&&((zr*zr+zi*zi)**0.5)<inf)
    zr_tmp = zr*zr-zi*zi+cr
    zi_tmp = zr*zi+zi*zr+ci

    zr = zr_tmp; zi = zi_tmp

    iter += 1
  end

  iter
}
```

図 1 Ruby で書かれたマンデルブロ集合を計算するプログラム

引数として n 個の整数値が与えられると，大きさ $x_1 \times x_2 \times \cdots \times x_n$ の n 次元配列生成する．初期化ブロック ({} 部分) が与えられた場合，各添え字値 $i_1, ..., i_n$ の下で e が GPU 上で並列に計算され，その値で要素値が初期化される．

- MDarray.new(a)
  引数として Ruby の配列 a が与えられると，その配列と同じ要素を持つ配列が GPU 上に生成される．Ruby の配列は行ごとに長さの異なる 2 次元配列のようなものも許されているが，a として与えることのできる配列は，それぞれの次元の長さが同じであり，要素の型もすべて整数型かすべて実数型でなければならない．

イテレータ

MDarray オブジェクトに対するイテレータには map と inject がある．

- a.map{|e| b }
- a.map_with_index {|$e, i_1, i_2, ..., i_n$| b }
  配列の各要素 e と添え字 $i_1, ..., i_n$ を用いて GPU 上で並列に b を計算し，その結果を要素とする a と同じ大きさの MDarray オブジェクトを生成する．式 b の中では，e と $i_1, ..., i_n$ のほかに，ブロックの外側で定義された変数を用いてもよい．
- a.inject(z){|x, y| b }
  a が 1 次元配列の場合，Ruby の Array#inject に相当する処理を GPU で並列に行う．具体的には a の全要素 (要素が不足する場合には一方を z として) をまず $(a[i], a[j])$ の対にして，$x = a[i], y = a[j]$ として GPU で b を並列に計算し，$v_0, ..., v_n$ を得る．これらから (x, y) の対を作り，同様の処理を行う．これを最終的に 1 要素になるまで繰り返す．Array# inject が添え字順に計算を行うのに対し，MDarray# inject は添え字の離れた要素どうしの計算を行うことがある．そのため，b は z を単位元とする可換な計算でなければならない．
  a の次元 n が 2 以上であった場合，n 次元目の配列それぞれに MDarray# inject を行った結果からなる $n-1$ 次元配列を生成する．

CPU 上での処理

前述の map，inject 以外のメソッドが呼び出されると，配列の内容が CPU メモリに転送され，Ruby インタプリタによって逐次処理される．ただし，

- $a[x, y, ...]$=val
- $a[x, y, ...]$

という形の要素アクセスは Array クラスのそれらから拡張されている．拡張点は多次元配列に対応していることと，map，inject へのブロック中に現れた場合は，CPU への転送が行われないことである．例えば，a.each{|e| puts(e) } という文が実行された場合，a の要素値が (もし GPU メモリにあったら)CPU メモリへ転送され，それから Array#each によって逐次的に実行される．

5 実現方法

本節では，Ikra の具体的な実現方法について述べる．処理系は Ruby プログラムからカーネルを分離して CUDA コードを生成する変換器と，実行時系である MDarray クラスからなる (図 2).

プログラマが記述した Ruby コードが変換器に入力されると，変換器はカーネルの CUDA コードと，イテレータ呼出をその CUDA コードの呼出に置き換えた Ruby コードを出力する．その Ruby コードを Ruby インタプリタ上で実行すると，MDarray クラスを通じてカーネルのバイナリが呼び出される．また，MDarray クラスはデータ転送などのために，あらかじめ用意された CUDA バイナリを呼び出している．

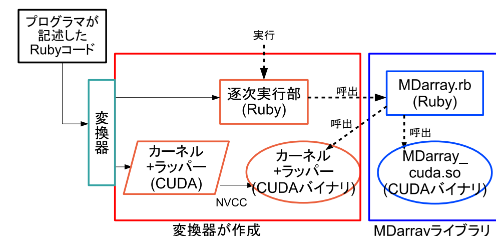

図 2 Ikra の実現方式の概観

5.1 変換器

変換器は CUDA コードを生成する部分と Ruby コードを生成する部分に大別される．生成された CUDA コードのカーネル部を図 3 に示す．

変換器は，まず Ruby プログラムを抽象構文木に変換する．次に GPU で並列実行するイテレータを Ruby の抽象構文木上で探し，イテレータブロックの部分木を取り出す．対象となるイテレータの特定については，5.3 節で述べる．次に，ブロック内で束縛されていない変数 (自由変数) を特定し，レシーバオブジェクト (_tmp) と，返値オブジェクト (_mda_0) および自由変数 (i_size, ..., r_min) を引数としてとる CUDA カーネル関数を生成する．生成されるカーネル関数に出現する変数の型情報は，動的型付言語である Ruby プログラム中には明示されていないため注意が必要である．これについては，5.3 節で述べる．カーネル関数の本体は，(1) イテレータブロック内で初めて束縛される変数を宣言する部分，(2) イテレータブロックの | | 中の変数に代入する部分，(3) イテレータブロック中の文を C 言語の文法に合わせて出力する部分，(4) イテレータブロックの返値を返値用配列に代入する部分の 4 つの部分からなる．

```
__global__ void __kernel_0(int x, int y ,
    float* _tmp, float* _mda_0 , int i_size ,
    int limit , float i_min , int inf , float
    res , int r_size , float r_min){

  //(1)カーネル内で初めて束縛される変数を宣言
  float zr; int iter; float zi; float zr_tmp;
  float zi_tmp; float cr; float ci;

  //(2)イテレータブロックの||中の変数
  int i = threadIdx.x+blockDim.x*blockIdx.x;
  int j = threadIdx.y+blockDim.y*blockIdx.y;
  int tid = i*x+j;
  float e = _tmp[tid]

  if(i<x && j<y){
    //(3)ブロック本体
    cr = (r_min + (res * i));
    ci = (i_min + (res * j));
    iter = 0; zr = 0.0; zi = 0.0;
    while(((iter < limit) && (pow(((zr * zr) +
          (zi * zi)), (float)0.5) < inf))){
      zr_tmp = (((zr * zr) - (zi * zi)) + cr);
      zi_tmp = (((zr * zi) + (zi * zr)) + ci);
      zr = zr_tmp; zi = zi_tmp;
      iter = (iter + 1);
    }
    //(4)返値を返値用配列に入れる
    _mda_0[i * x + j] = iter;
  }
}
```

図 3 生成された CUDA カーネル

```
result = MDarray.new(r_size , i_size).
    gmap_wi_with_kernel(_kernel.method("
    _kernel_0") ,[res , limit , inf , r_min ,
    i_min])
```

図 4 生成されたラッパー呼出部

また，変換器は CUDA カーネルを Ruby から呼び出すためのラッパーも生成している．

生成される Ruby コードは，変換対象のイテレータ呼出を上記のラッパーの呼出で置換したコードである．例えば図 1 の 6 行目は図 4 のようなコードに置換される．図 4 のコードは，呼び出すカーネルの種類 (map_with_index など) に応じたカーネル呼出メソッド gmap_wi_with_kernel と，その引数からなる．引数は，第 1 引数としてカーネルを呼び出すラッパーメソッド，第 2 引数としてカーネルに渡される自由変数 (res, ..., i_min) の配列からなる．

5.2 MDarray

並列データクラスである MDarray は，メモリ管理やラッパー呼出のためのメソッドを持ち，Ruby と CUDA で記述されている．

MDarray の要素データは CPU メモリと GPU メモリのどちらかに置かれる．そのため，どちらのメモリ上のデータが最新かを記憶するフラグを持つ．そして，MDarray クラスのメソッド呼出の際に，必要に応じてデータ転送を行う．例えば GPU で実行される CUDA カーネルを呼び出す際に，CPU メモリ上のデータが最新の場合にはデータ転送を行い，GPU メモリ上のデータが最新の場合には行われない．

また，Ruby のゴミ集め機構に連動させることで，GPU 側メモリ上のデータの自動的な解放を行う．具体的には，MDarray データが最初に GPU メモリに転送されるときに，その領域解放用のファイナライザを MDarray オブジェクトに ObjectSpace#define_finalizer を用いて登録している．また，GPU メモリの割当に失敗した時は，Ruby のゴミ集めを起動する．これによって，Ruby オブジェクトとしての MDarray オブジェクトの消滅時に GPU メモリの開放を実現している．

5.3 現状

現在までに，簡単なプログラムを GPU で実行できるプロトタイプ処理系が完成している．処理系は Ruby と CUDA で書かれ，その行数は表 2 のとおりである．Ruby プログラムの構文解析には ruby_parser を用いた．前節までに紹介した機能のうち，制限されているものや，実現されていないものは以下のとおりである．

変換器

変換器が型情報を得るためには，Ruby プログラム中に型宣言を挿入させることで対処している．そのための疑似メソッドとして，変数 $var_1, var_2, ...$ の型が float 型であることを宣言する decl_float($var_1, var_2, ...$) などを用意している．

MDarray に対するイテレータとしては，2 次元配列に対する map と 1 次元配列に対する inject のみが実現されている．現在は，カーネルをイテレータ名によって特定するため，それぞれ Array クラスのメソッドとは異なる gmap_with_index2，ginject という名前になっている．また，イテレータブロック内に記述できる構文としては，四則演算やべき乗などの演算，条件分岐の if 文，繰り返し処理の while 文，メソッド呼出などがある．

表 2 Ikra 処理系の行数

変換器	1600 行 (Ruby)
MDarray	400 行 (Ruby)
	300 行 (CUDA)

MDarray

MDarray は多次元配列として設計されているが，現在は 2 次元までの単精度浮動小数点数のみを許す．また，Array クラスに用意された each などのメソッドは実現されていないため，明示的に Ruby の配列に変換した後で実行しなければならないという制約がある．

6 性能評価

Ikra を用いて，GPGPU プログラムの大きさ，GPU 実行の速度について評価を行う．評価環境は表 3 のとおりである．

GPGPU プログラムとしては，400 万要素に対するマンデルブロ集合の計算と 1600 万要素の総和計算 (図 5) をもちいた．それぞれのプログラムの GPU 実行版を Ikra および CUDA で，CPU 実行版を Ruby および C で作成し，カーネル部とデータ転送の時間[†4]を測定した．CUDA 版は，Ikra が生成する CUDA コードと同程度の最適化[†5]を行ったものを用いた．また，それぞれのプログラムのコードの行数を比較した．その結果を表 4，5 に示す[†6]．

Ikra による実行は，カーネル部の CUDA との比較で 0.9〜1 倍程度，Ruby と比べると 2000〜8000 倍の性能を得られている．カーネルと転送の時間の合計

表 3 実行環境

CPU	Core2Quad Q8300 2.5GHz
メインメモリ	2GB
GPU	GeForce GTX 465(コア 607MHz，SP 1215MHz)
ビデオメモリ	1GB
接続バス	PCI-Express x16
OS	Debian 4.1(kernel 2.6.26-2-686)
コンパイラなど	ruby1.8.7-p334 gcc4.1.3 -O3 nvcc3.1v0.2.1221 CUDA Driver Version 4.0

†4 マンデルブロ集合の計算では，CPU メモリから GPU メモリへのデータ転送は行われない．

†5 CUDA 版では，さらなる最適化が可能である．例えば総和計算では，for 文のアンロールなどを行うことにより数倍の高速化が可能である．

†6 それぞれ 10 回計測を行った平均である．Ikra，Ruby では Time オブジェクトを用いて，C では gettimeofday 関数により，CUDA では CUDA イベント (cudaEventCreate など) により時間を測定した．

```
require "MDarray"

n = 1024*1024*4
ary = Array.new(n){rand}

mda = MDarray.new(ary)
gpu_result = mda.ginject(0.0){|x,y|
  decl_float(x, y)
  x + y
}
```

図 5　総和計算のコード

表 4　マンデルブロ集合の計算時間及び行数

	Ikra	CUDA	Ruby	C
カーネル (ミリ秒)	19	7	68000	1915
転送 (GPU → CPU) (ミリ秒)	32	24	-	-
行数 (行)	50	65	25	60

表 5　総和計算の時間及び行数

	Ikra	CUDA	Ruby	C
転送 (CPU → GPU) (ミリ秒)	44	44	-	-
カーネル (ミリ秒)	9	9	17700	102
転送 (GPU → CPU) (ミリ秒)	<1	<1	-	-
行数 (行)	10	105	5	25

でも，Ikra は CUDA と同程度，Ruby の 300〜2000 倍程度の性能が得られている．カーネルの実行時間と転送の時間を比べると転送により多くの時間を費やしているが，複数のカーネルを繰り返す場合はカーネル間で転送は行われないため，転送の時間は問題とならない．

プログラマの書くコードの行数は，記述可能な構文の制限や型宣言により Ruby と比べると増加しているものの，CUDA よりは少なくなっている．これらのことから，少なくとも今回示したような簡単な GPGPU プログラムに関しては，Ikra を用いることで容易に作成できると言える．

7　関連研究

7.1　CUDA を用いる処理系

簡単に GPGPU プログラミングを行うことを目的とした処理系は多数存在しており [2] [7] [11] [1] [4] [9]，それらは 2 節で述べたとおりである．

7.2　CPU での並列実行

複数 CPU を用いた共有メモリ型計算機における並列実行環境としては，C にプラグマを挿入することで並列化を行う OpenMP [8] などがある．OpenMP は逐次実行部と並列実行部を繰り返して処理を行う点では Ikra と同じであるが，メモリ間のデータ転送が存在しない点が Ikra とは異なる．

7.3　Ruby の高速化

高速化を目的に Java で実現された Ruby インタプリタとして，JRuby [3] がある．また，画像処理を対象に配列の計算を高速化した Ruby 処理系として，HornetsEye [10] がある．これらはいずれも言語を変換しているという点で Ikra と共通点があるが，GPU 実行のための追加情報も出力される点が Ikra とは異なる．これらの変換手法を参考とすることで，Ikra のコード変換を改良することが考えられる．

8　まとめと今後の課題

本研究では，GPU 上で実行できる Ruby プログラムを作成させるために，Ikra 処理系として並列データクラスである MDarray と言語変換器のプロトタイプを作成した．この処理系を用いて，マンデルブロ集合計算と総和計算のプログラムを実行した結果，CUDA よりも少ない行数で CUDA に近い実行速度を得ることができた．

今後の課題としては，処理系の完成度を高め，多くの応用例について実験することがある．また，プログラマが型情報を与えずとも CUDA コードに変換できるように型推論を行うこと，あるいは実行時情報を用いてコード生成とコンパイルを行うことが考えられる．さらに，MDarray のとることのできる要素型についても制限を緩和し，オブジェクトをとることが考えられる．最適化に関しては，プログラマがブロック数などを指定できるようにすること，共有メモリを明示的に利用できるようにすることなどが考えられる．

謝辞
本稿の作成にあたり，櫻井孝平氏，渡邉卓也氏，当山

学氏，玉井哲雄氏，紙名哲生氏より有益なコメントをいただいたことに感謝します.

参 考 文 献

[1] Chakravarty, M.M.T. Keller, G. Lee, S. McDonell, T.L. and Grover, V.:Accelerating Haskell array codes with multicore GPUs, Proc. of the sixth workshop on Declarative Aspects of Multicore Programming, DAMP '11, pp. 3–14.
[2] Han, T.D. and Abdelrahman, T.S.:hiCUDA: a high-level directivebased language for GPU programming, Proc. 2nd Workshop on General Purpose Processing on Graphics Processing Units, ACM, 2009, pp. 52–61.
[3] JRuby, http://www.jruby.org/, 2011/8/8 アクセス
[4] Klöckner, A. Pinto, N. Lee, Y. Catanzaro, B. Ivanov, P. and Fasih, A.:PyCUDA: GPU run-time code generation for high-performance computing, 2009, Submitted to Parallel Computing, Elsevier, arXiv:0911.3456v2 [cs.DC]
[5] Munshi, A.:The OpenCL Specification , 2009
[6] NVIDIA:NVIDIA CUDA C Programming Guide Version 4.0, 2011.
[7] 大島聡史, 平澤将一, 本多弘樹:OMPCUDA : GPU 向け OpenMP の実装, 情報処理学会研究報告. [ハイパフォーマンスコンピューティング] 2008(125), pp. 121–126.
[8] OpenMP Architecture Review Board:OpenMP specification v3.1, 2011.
[9] SGC Ruby CUDA, http://rubyforge.org/projects/rubycuda, 2011/8/8 アクセス
[10] Wedekind, J. Amavasai, B. Dutton, K. and Boissenin, M.:A machine vision extension for the Ruby programming language, Int. Conf. on Information and Automation, 2008, pp. 991–996.
[11] Yan, Y. Grossman, M. and Sarkar, V.:JCUDA: A Programmer-Friendly Interface for Accelerating Java Programs with CUDA, Proc. of the 15th International Euro-Par Conference on Parallel Processing, 2009.