Vol.2010-HPC-126 No.8
2010/8/3

プログラミング言語MLのCUDA向け拡張

野瀬貴史[†1]　平木敬[†1]

CUDA, BrookGPU をはじめとした GPGPU のための開発環境は，その多くが C 言語の拡張として提供されている．これは，C や C++で書かれたコード資産を少ない改造で GPGPU に対応させる状況を考えれば，妥当である．しかし，C 言語ベースの開発は，開発効率において現代的な言語に比べれば低い．また，CUDA においては最適化を手動に頼らなくてはならない場面があり，煩雑である．そこで，本研究では，現代的な言語の一つである ML で書かれたプログラムから，CUDA 向けのプログラムを出力するコンパイラを作成し，開発効率の向上を目指す．本論文では，関数型言語 ML のサブセットを対象とした教育用コンパイラ MinCaml に拡張を加えた CUDA コンパイラを実装し，アプリケーションに必要なコード量と性能を評価した．

An Extension of Programming Language ML for CUDA

Takafumi Nose[†1] and Kei Hiraki[†1]

Most GPGPU development environments are often provided as an extension of Programming Language C. These are reasonable solutions in the case of making existing codes, that is written in C or C++, to be accelerated by GPGPU with small efforts. But, productivity of development based on C or its extension is lesser than modern language, and its optimization must be performed by hand in the case of CUDA. In this work, we develop a compiler for one of the modern languages, ML, that emits programs for CUDA. It aims to solve the complexity of C-based programming. In this paper, we implemented a compiler for CUDA that is based on MinCaml, that is a ML subset compiler for educational purposes, and evaluated it.

†1 東京大学
The University of Tokyo

1. はじめに

数値計算をする際は，パフォーマンスが優先される場面が多いため，プログラムを書く際の言語選択は C か Fortran となるのが通例である．また，パフォーマンス向上のために，特定の計算を速くするハードウェアアクセラレーションをシステムに付け足すことは，古くはコプロセッサとしての FPU から，現在の GPGPU に至るまで，広く行われている．

C 言語・Fortran は 30 年以上の歴史があり，コード資産が多い．よって，既存のコード資産を少ない手間で加速したいという状況を考慮すれば，ハードウェアアクセラレーションのためのソフトウェア記述が C 言語の拡張（あるいは Fortran の拡張[1)]）として提供されるのは妥当である．

実際，GPU のプログラマブルシェーダの発展に伴い登場した Cg (C for graphics)[2)], GLSL (OpenGL Shading Language)[3)], HLSL (Microsoft ' s High-Level Shading Language)[4)] といったシェーディング言語は C 言語をベースとしており，初期の GPGPU は本来グラフィック処理を記述するためのこれらの言語を一般の計算に転用する形で実装されていた．次いで，より一般的な計算を記述できる言語として Sh[5)], Scout[6)], BrookGPU[7)] が登場し，ついには GPU メーカーが CUDA，Brook+を GPGPU 開発言語として公式にサポートするに至った．最近では，GPU に限らない多様な環境で並列計算を行うために，OpenCL としてこれらの言語は標準化されつつある．これらの C 言語拡張は，より使いやすく，多様な環境で動くように，という方向に進んできたと言える．

しかし，C 言語ベースの開発は，Java，C#などの C 言語の文法をまねた後発言語や，Python，Ruby といったスクリプト言語に比べて開発効率が低い．また，CUDA においてはシェアードメモリの活用，コアレッシングを考慮した最適化を手動に頼らなくてはならず，煩雑である．このため，C 言語と比較して開発効率に優れた言語で GPGPU を行うアプローチがいくつか存在する．

まず JCublas, JCufft[8)] のように，CUBLAS や CUFFT など，GPGPU に対応した数値計算ライブラリのラッパを書くという手段がまず存在する．BLAS 演算のような決まりきった演算しかしないのであれば，これで十分である．特に，CUBLAS のインターフェースは通常の C 言語であり，C ライブラリを呼び出せない実用的な言語は少ないため，ラッパの実装の労力は小さい．しかし，このアプローチでは新たに GPU カーネルを定義することは出来ず，柔軟性に欠ける．

CUDA の API は C 言語ベースであるが，CUDA の nvcc コンパイラは C++をコンパ

イルできる．このため，C++のライブラリを整備することによって高級な使い方をすることができる．Thrust[9] はその一例である．CUDA 向けの C++テンプレートライブラリであり，STL と似たインターフェースを備えている．CPU と GPU 間のデータ転送に関わる煩雑さが隠蔽されており，プログラミングモデルは関数型言語と似た部分がある．しかし，C++では，関数オブジェクトは operator() をオーバーロードしたクラスを定義することにより実現されるので，不必要に煩雑な部分がある．

別の手段として，高級言語で書かれたプログラムを GPU で動かせるようにコンパイルするというアプローチがある．PyGPU[10][11] は，Python 向けのドメイン固有言語であり，画像処理を GPU で行うことを目的としている．目的が特化されており，Cg 環境が前提となるため，CUDA による一般的なプログラムの開発には現状では使えない．

高級言語を既存の低級な言語環境向けに変換するアプローチの例として Vala[12] がある．これは，通常 C 言語で行われる GObject 型システムをベースとした開発を楽にできるようにすることを目的とした言語である．文法は C#に似ており，コンパイル後は C 言語のソースコードが出力される．これにより，C 言語の API と ABI を保持したまま，実行時に余計なランタイムを必要としない軽量で高速なプログラムを，現代的な文法の言語で開発できる．この方式の利点は，C 言語拡張の多い GPGPU 環境にもそのまま適合すると考えられる．

GPGPU のプログラミングモデルは，GPU カーネルに配列を渡すと，各スレッドが配列の担当部分を処理するというものであり，関数型言語で多用される map 処理と親和性が高い．また，処理内容は出来る限り副作用が少ないものが望ましい．よって，CUDA への変換のもととなる高級言語は関数型言語か，関数型言語のパラダイムを取り入れた言語が適している．

そこで，本研究では，現代的な関数型言語のプログラムから，GPGPU 向けの C 言語ソースを出力するコンパイラを作成し，開発効率の向上を目指す．本研究では，言語には ML を選択した．プログラミング言語 ML は型推論を備えた非純粋・正格評価・静的型付けな関数型言語であり，関数型ではなく命令的な記述も可能という特徴を持っている．Haskell と違い，プログラムを全面的に純粋関数型言語のパラダイムで書く事をユーザに強要しないため，既存の手続き型言語に慣れたユーザでも気軽に関数型言語でのプログラミングに取り組みはじめることができる．また，Microsoft 社の Visual Studio はバージョン 2010 から ML の方言である F#をサポートしており[13]，今後 ML 派生言語の一般への広まりが期待できる．本論文の実装は，ML の一方言 OCaml のサブセットの処理系である MinCaml[14] をベースとしている．対象とする GPGPU 環境は CUDA とした．

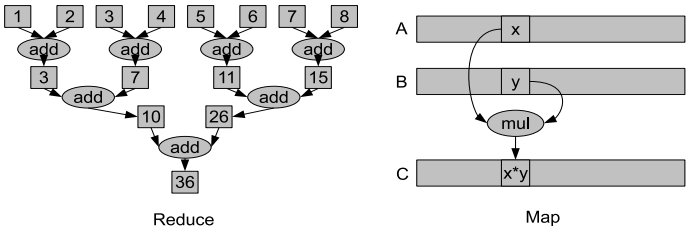

図 1 Reduce 演算と Map 演算

本論文は以下の構成となっている．第 2 章では，本論文のコンパイラの設計について述べる．第 3 章では，実装の現状について報告する．第 4 章では，今回の実装を評価する．第 5 章では，関連研究と本研究との関係を述べる．第 6 章では，本論文をまとめ，今後の課題について述べる．

2. 設　　計

本論文でベースとした MinCaml は合計で 2000 行程度のコンパクトなコンパイラであり，ユーザによる改造が容易である．このため，言語拡張の試験実装に適している．MinCaml が入力とする ML のサブセットはパターンマッチや代数的データ型が欠如しており，ML としては重要な機能の欠如[15] だが，代数的データ型から値を取り出すことを主眼とする以外のパターンマッチは条件分岐を含み，GPU カーネルの記述には向かないため，試験実装では不要と判断した．

2.1 追 加 要 素

PArray　GPU 側で保持する配列を表すデータ型．現在の設計では，int 型と float 型のみを保持できることにした．MinCaml の Array 型は破壊的な代入をすることが可能だが，PArray ではすることができない．

map, map!　ある関数と，その関数の引数と同じ数の int 型，float 型または PArray 型の引数を取り，map 演算（**図 1** 右）を行う命令．map!は，map の破壊的バージョンである．OCaml においては，引数一つを取る関数と，配列一つのみを取る Array.map : ('a -> 'b) -> 'a array -> 'b array という同種の機能が存在する．

reduce　PArray に対し，引数を二つ取る関数を作用させ，reduce 演算（図 1 左）を行う命令．OCaml においては，Array.fold_left, Array.fold_right という似たような機能が

```
(* x, y は float を格納する PArray *)
let xplusy = map add x y in (* (1) *)
let xplus3 = map add x 3 (* (2) *)
```

図 2 map の際 add に渡す引数はスカラ値でも PArray でも構わない

存在する．

PMatrix, Matrix GPU 側で行列を保持するデータ型 PMatrix と，それと対応する CPU 側のデータ型 Matrix.

PGrid, Grid, PSpace, Space Grid は 2 次元の，Space は 3 次元の Array である．PGrid, PSpace はそれぞれ対応する GPU 側のデータ型である．

3. 実　　装

map の実装は，map する関数に渡すデータは PArray であっても，スカラ値であっても構わないようになっている．つまり，let rec add x y = x +. y という関数があるとき，**図 2** の (1), (2) の両方の記述が許容され，出力される C++のソースは**図 3** のようになる．これにより，ベクトル-ベクトル演算とベクトル-スカラ演算が，共通の関数一つを定義するだけで実現できる．map, map!, reduce に渡される関数はデバイス側で呼び出される関数と判断され，自動で関数識別子__device__が付加される．PMatrix の実装は，CUBLAS のラッパにとどまっている．GPU カーネル内での GPU に特化した自動的な最適化は，現在はまだ実装されていない．

4. 評　　価

4.1 コード量

SAXPY の ML，C 言語，Thrust を用いた C++での記述を**図 4**，**図 5**，**図 6** に示す．C 言語で書いた場合，GPU カーネルの記述は単純にはならず，メモリの確保や解放をその都度明示しないといけないため，煩雑である．Thrust では，メモリ周りの操作がラップされており，かつ関数型言語における map と同じプログラミングスタイルで記述できるが，関数オブジェクトの記述では，本質的には不要な struct や operator() などの記述が必要となる．ML で書いた場合，GPU カーネルは自動的に判別されるため，map される関数に特別な記述は不要である．

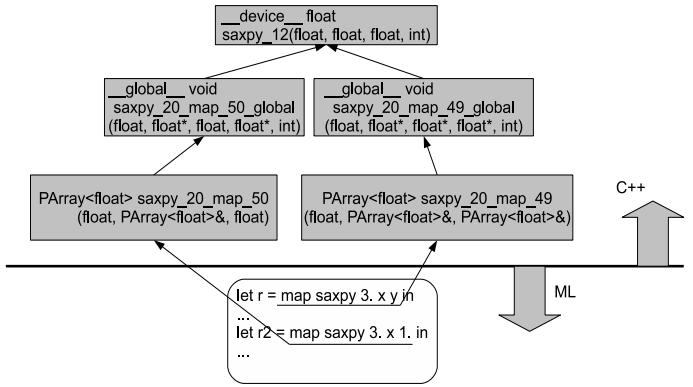

図 3 型の違う引数の map 演算があるときの C++ソース

```
let rec saxpy a x y =
  a *. x +. y in
let x = PArray.create 128 1. in
let y = PArray.create 128 2. in
let r = map saxpy 3. x y
```

図 4 GPU による SAXPY を ML で書いた例

4.2 コンパイル後のコード

コンパイル後のコードのオーバーヘッドがどの程度か調べるために，ML からコンパイルしたソースコードと手で書いた C・Thrust を用いた C++で同等の処理を記述し，500000 回の SAXPY を行うベンチマークを行った．実行環境の CPU は Intel Core 2 Quad Q6700 2.66GHz, GPU は NVIDIA Tesla C1060, OS は CentOS 5.4 であった．ベンチマーク結果とコードサイズは表 1 である．結果，手で書いた C と大差ない性能のコードが出力されていることが確認された．Thrust が ML, C の 4 倍の実行時間となっているが，このうち実際に計算を行っているのは 20sec 程度であり，メモリ確保・解放処理のオーバーヘッドが大きいことが確認された．

```
__global__ void fill(float a, float* x, int N) {
    int tid = blockIdx.x * blockDim.x + threadIdx.x;
    if(tid < N) x[tid] = a;
}
__global__ void saxpy(float a, float* x, float* y, float* r, int N) {
    int tid = blockIdx.x * blockDim.x + threadIdx.x;
    if(tid < N) r[tid] = a*x[tid]+y[tid];
}
void do_saxpy_plainc() {
    int *x, *y, *r;
    dim3 grid(1,1); dim3 block(128,1,1);
    cudaMalloc((void**)&x, sizeof(int)*128);
    cudaMalloc((void**)&y, sizeof(int)*128);
    cudaMalloc((void**)&r, sizeof(int)*128);
    fill<<<grid, block>>>(1.0f, x, 128);
    fill<<<grid, block>>>(2.0f, x, 128);
    saxpy<<<grid, block>>>(3.0f, x, y, r, 128);
    cudaFree(x); cudaFree(y); cudaFree(r);
}
```

図 5 SAXPY を C で書いた例.

5. 関 連 研 究

5.1 Haskell ベースの研究

高級言語から低レベルな並列化されたコードを生成する研究の多くが Haskell を拡張することによりなされている．Lee らの Haskell 拡張[17] や Warp Speed Haskell[18] は，GPU や並列環境向けのデータ型を定義し，それ専用の map, zipWith といった操作を加えるものである．17) にはオンラインコードジェネレータがあり，必要に応じて CUDA の C 言語ソースを生成し，CUDA コンパイラでコンパイルしたのちに動的にロードするという機能を備えている．本研究では動的なコンパイルは対象にしていないものの，FFTW にふくまれる genfft コンパイラ[19] のようなオートチューニングを支援する機構を検討中である．

表 1 SAXPY の動作速度とコードサイズ

	ML	C	Thrust
SAXPY 実行時間 (sec)	90.95	90.70	362.12
コードサイズ (bytes)	28085	18556	147172

```
struct saxpy_functor {
    const float a;
    saxpy_functor(float _a) : a(_a) {}
    __host__ __device__
        float operator()(const float& x, const float& y) const {
            return a * x + y;
        }
};
void do_saxpy_thrust() {
    thrust::device_vector<float> X(128, 1.0f), Y(128, 2.0f), R(128);
    float A = 3.0f;
    thrust::transform(X.begin(), X.end(), Y.begin(),
                      R.begin(), saxpy_functor(A));
    return;
}
```

図 6 SAXPY を Thrust を使った C++で書いた例.
saxpy_functor の実装は 16) による.

Obsidian[20] は，同じく Haskell に組み込まれたハードウェア記述言語 Lava[21] の影響を受けている．両者とも，”>->” または ”->-”という記号を用いて，回路をつなぎ合わせるようなプログラミングスタイルを取る．17) に比べると粒度の細かい記述が可能である．データフロープログラミングを行うには向いているが，一般のプログラムに必ずしも使えるスタイルではない．

Ypnos[22] は，計算流体力学によく現れる構造化されたグリッドに関する計算を対象としている．グリッド中の注目している点の周りのステンシルを宣言的かつ視覚的に定義できるのが特徴である．これにより記述しやすくなる問題としては，ラプラス方程式，ライフゲーム，ガウス-ザイデル法がある．また，境界条件の記述のため，lift というプリミティブが提

案されている．これは，有限のグリッドと境界条件から無限のグリッドを定義するものである．グリッドに適用する関数に対して，空間が無限であるように見せかけられるため，境界条件を意識せずに書けるようになっている．22) では紙面の都合により lift の詳細な説明は省かれている．Orchard は今後の課題として，場所によってメッシュの粒度が違うグリッドへの対応を検討している．我々研究の設計に含まれている PGrid，PSpace は Ypnos と似た記法の導入のために入っているが，3 次元以上のステンシルの記述は困難であるため，何らかの支援エディタが必要であると考えている．

5.2 Intel Ct

Intel Ct (C for throughput computing)[23] は，C/C++をベースとして，TVEC というテンプレートベースの並列コンテナとそれに対する操作を導入した言語である．ソースコードは中間言語に変換され，動作環境にあわせて JIT コンパイルされる．TVEC はネストが可能である．また，配列中の注目点からの相対アクセスのために，TElt という型が導入されている．

5.3 OCaml

OCaml では，CUDA の API に対するラッパを書くアプローチとして Daml[24] という個人のオープンソースプロジェクトが存在するが，メモリの通信とデバイス情報取得の API をラップする段階でとどまっており，GPU カーネルは記述できない．また，2008 年 12 月を最後に更新されていない．我々の知る限り，このプロジェクトのほかに ML 派生言語で GPGPU を行う試みは存在しない．

OCaml から別の言語へのトランスレータの一つに，OCamlJS[25] がある．Web アプリケーションや Firefox 拡張機能の開発に JavaScript ではなく OCaml を使うためのトランスレータであり，JavaScript では実現できないコンパイル時の型チェックや OCaml 周辺ツールとの協調を可能にしている．高性能ではなく，開発効率を第一の目的に据えている点で本研究とは異なる．

6. まとめと今後の課題

本研究では，GPGPU における C 言語ベースの開発の煩雑さの軽減のため，CUDA 向けのプログラムを出力する ML サブセットのコンパイラを作成した．関数型言語のプログラミングモデルが自然に GPGPU に適用でき，C/C++で書いた場合よりも少ない労力で記述できることを示した．また，ML のソースから生成後のソースコードと，手で書いた同等の C 言語のソースを比べたとき，顕著なオーバーヘッドは見られないことを示した．

今回の実装では配列に対し map する関数，つまりコンパイル後は GPU カーネルとなる関数内の最適化は考慮していない．GPGPU プログラミングにおけるもうひとつの困難である最適化の自動化は今後解決しなければならない課題である．

現在の PArray は int と float しか格納することが出来ない．同じ長さの配列を多数持つより，tuple を格納した配列が一つだけの方が書きやすいケースがあるし，多次元の空間を表すのに PArray をネストして使いたい場面もあるため，より複雑なデータ構造とそれに対する演算への対応を進めたい．

現在の PMatrix は CUBLAS のラッパに過ぎず，それを呼び出す上での最適化はなされていない．実際の数値計算では疎行列がよく現れるが，CUBLAS は疎行列に特化した機能を持たない．行列が密行列か疎行列か推論し，最適な格納方式を選択する方法を考えていきたい．

複数 GPU を使うことやデータの転送の隠蔽は，性能を出す上で重要である．また，Fermi 世代の GPU は複数カーネルを同時実行できる機能を持つ．これらは遅延評価の導入によって自然に記述できると考えられるので，遅延評価の実装を行ないたい．

今回の研究で施した拡張は，CUDA に強く依存するものではないため，OpenCL や OpenMP といった環境向けのポーティングが容易である．需要があれば取り組んでみたい．

参 考 文 献

1) STMicroelectronics: PGI Accelerator Compilers, http://www.pgroup.com/resources/accel.htm.
2) Fernando, R. and Kilgard, M.: *The Cg Tutorial: The definitive guide to programmable real-time graphics*, Addison-Wesley Longman Publishing Co., Inc. Boston, MA, USA (2003).
3) Rost, R.: *OpenGL (R) Shading Language*, Addison Wesley Longman Publishing Co., Inc. Redwood City, CA, USA (2004).
4) Gray, K.: *Microsoft DirectX 9 programmable graphics pipeline*, Microsoft Press (2003).
5) McCool, M., Qin, Z. and Popa, T.: Shader metaprogramming, *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS conference on Graphics hardware*, Eurographics Association, p.68 (2002).
6) McCormick, P., Inman, J., Ahrens, J., Hansen, C. and Roth, G.: Scout: A hardware-accelerated system for quantitatively driven visualization and analysis, *Proceedings of the conference on Visualization'04*, IEEE Computer Society, p.178 (2004).

Vol.2010-HPC-126 No.8
2010/8/3

7) Buck, I., Foley, T., Horn, D., Sugerman, J., Hanrahan, P., Houston, M. and Fatahalian, K.: BrookGPU web site (2003).
8) Hutter, M.: Java bindings for CUDA, http://www.jcuda.org/.
9) Bell, N.: Thrust, http://code.google.com/p/thrust/.
10) Lejdfors, C. and Ohlsson, L.: Implementing an embedded GPU language by combining translation and generation, *SAC*, Vol.6, Citeseer, pp.1610–1614.
11) Lejdfors, C. and Ohlsson, L.: PyGPU: A high-level language for high-speed image processing (2007).
12) Billeter, J and Sandrini, R.: Vala - Compiler for the GObject type system, http://live.gnome.org/Vala.
13) Mackey, A.: *Introducing. Net 4.0: With Visual Studio 2010*, Apress (2009).
14) Sumii, E.: MinCaml: a simple and efficient compiler for a minimal functional language, *Proceedings of the 2005 workshop on Functional and declarative programming in education*, ACM, p.38 (2005).
15) 住井英二郎：MinCaml コンパイラ，コンピュータ ソフトウェア， Vol.25, No.2, pp. 2–2 (2008).
16) Bell, N.: QuickStartGuide - thrust - A brief tutorial for new Thrust developers, http://code.google.com/p/thrust/wiki/QuickStartGuide.
17) Lee, S., Chakravarty, M., Grover, V. and Keller, G.: GPU kernels as data-parallel array computations in Haskell, *Workshop on Exploiting Parallelism using GPUs and other Hardware-Assisted Methods*, Citeseer (2009).
18) Jones, W.: Warp Speed Haskell (2009).
19) Frigo, M. and Kral, S.: The Advanced FFT Program Generator GENFFT, *Aurora Technical Report TR2001-03*, Vol.3 (2001).
20) Svensson, J. and Göteborg, S.: An embedded language for data-parallel programming (2008).
21) Bjesse, P., Claessen, K., Sheeran, M. and Singh, S.: Lava: hardware design in Haskell, *Proceedings of the third ACM SIGPLAN international conference on Functional programming*, ACM, pp.174–184 (1998).
22) Orchard, D., Bolingbroke, M. and Mycroft, A.: Ypnos: declarative, parallel structured grid programming, *Proceedings of the 5th ACM SIGPLAN workshop on Declarative aspects of multicore programming*, ACM, pp.15–24 (2010).
23) Intel Corporation: Ct Technology, http://software.intel.com/en-us/data-parallel/.
24) Govender, S.: daml - an OCaml binding to CUDA, https://forge.ocamlcore.org/projects/daml/.
25) Donham, J: From OCaml to Javascript at Skydeck, http://cufp.org/archive/2008/slides/DonhamJake.pdf.