Hastega: Elixirプログラミングにおける超並列化を実現するためのGPGPU活用手法

山崎 進[1,a)] 森 正和[2,b)] 上野 嘉大[2,c)] 高瀬 英希[3,d)]

概要：Elixir では Flow という MapReduce に近いモデルの並列プログラミングライブラリが広く普及している．Flow を用いるとパイプライン演算子を用いた簡潔な表現でマルチコア CPU の並列性を活用することができる．我々は Flow によるプログラム記述が GPGPU にも容易に適用できるという着想を得て，OpenCL によるプロトタイプを実装した．現行の GPU の一般的なアーキテクチャである SIMD では，単純な構造で均質で大量にあるデータを同じような命令列で処理する場合に効果を発揮する．一方，Flow では，単純な構造で均質で大量にあるデータであるリスト構造に対し，パイプライン演算子でつながれた一連の命令列で処理する．そこで，この一連の命令列をひとまとめにして GPU 向けにコンパイルし，入力となるリスト構造を配列データにまとめてコードとともに GPU に転送して実行することで高速化を図る手法，Hastega を提案する．そこで，素体のロジスティック写像を用いたベンチマークプログラムを開発し，期待される性能向上がどのくらいになるかを評価した．Mac Pro と Google Compute Engine (GCE) で評価した．言語は Elixir，Elixir から Rust によるネイティブコードを呼び出したもの (Hastega)，Rust によるネイティブコード，Python で比較した．その結果，次の 3 つの結果が得られた: (1) GPU を使用した Rustler のコードは，Elixir 単体のコードと比べて 4.43〜8.23 倍高速になった (2) GPU を使用した Rustler のコードは，GPU を使用するネイティブコードと比べて，1.48〜1.54 倍程度遅くなっただけである (3) GPU を使用した Rustler のコードは，GPU を使用する Python のコードと比べて，3.67 倍高速である．今後，本研究で得た知見を元に LLVM を用いてコード生成器を含む処理系を開発する予定である．

キーワード：Elixir，GPGPU，並列プログラミング言語，並列計算，MapReduce

Hastega: A Method Using GPGPU for Super-Parallelization in Elixir Programming

Yamazaki Susumu[1,a)] Mori Masakazu[2,b)] Ueno Yoshihiro[2,c)] TAKASE Hideki[3,d)]

Abstract: Elixir has Flow that is a popular parallel programming library similar to MapReduce. Flow can realize parallel programming on multi-core CPUs by simple description using pipeline operators. We ideate that code description of Flow can be applied to GPGPU easily, and implements prototypes. The SIMD architecture that current general GPUs adopt is effective when a single instruction sequence processes a simple, homogeneous and mass data structure, and code using Flow is a single instruction sequence connected by pipeline operators that processes a linked-list structure, which is asimple, homogeneous and mass data structure. Thus, we propose the Hastega method, which is a code optimization method by compiling the instruction sequence for the GPU, sending it and a mass array composed of data from the linked-list, and executing them. We develop a benchmark suit of the logistic maps over prime fields, and evaluate performance of the proposal system using it. We evaluate it using Mac Pro and Google Compute Engine (GCE). We compare Elixir, Rustler, which is Elixir that calls native code by Rust, native code by Rust and Python. We get the following results: (1) Elixir and Rustler code using GPU (Hastega) is 4.43–8.23 times and 7.43–9.64 times faster than pure Elixir and Python code executed by only CPU, respectively. (2) The performance gap of Elixir and Rustler code using GPU by our strategy is only 1.48–1.54 times compared with native code using GPU. (3) Additionally, our Hastega method for Elixir achieves 3.67 times faster than Python code with GPU. We plan to develop a processing system including a code generator using LLVM based this research.

Keywords: Elixir, GPGPU, Parallel programming languages, Parallel computing, MapReduce.

1. はじめに

GPGPU 技術 [10] が活発に研究開発されてきた．現在主流となっている GPU のアーキテクチャは SIMD(単一命令複数データ) モデルに基づいていて超並列計算に向いているため，システムを並列に計算させることができる．

GPGPU の研究開発が注目されてきた理由の 1 つは，機械学習が社会実装に適用されることが一般的になり，より高いパフォーマンスが要求されるようになってきたからである．機械学習で用いられるプログラミング言語のデファクトスタンダードは Python [20] である．しかし，CUDA [11] や CuPy [14]，TensorFlow [6]，Keras [2] といった Python ベースの現行の GPGPU ライブラリには次の 2 つの問題点がある．

1. **性能の問題:** Python [20] は，NumPy [12] や CuPy [14] のようなネイティブライブラリを除けば単一スレッドのインタプリタ言語であるため，基本的に低いパフォーマンスである．このことにより，システムがネイティブライブラリで処理する前に Python インタプリタで大量のデータを処理するときに深刻な問題となる．
2. **設定の問題:** Python が GPU を使えるようにするネイティブライブラリを設定するのが難しい．これらはインストールや設定がとても面倒である．さらに，NVIDIA の GPU しかサポートされない．

このことから，我々は機械学習の目的で Elixir [18] を採用する．その理由は，Elixir はとても良い並列処理性能を発揮するからである [4]．しかし，現在の Elixir は GPGPU をサポートしない．そこで，我々は Elixir に GPGPU サポートを実装することを試みた．

我々が提案する Hastega は，MapReduce モデル [3] に似ている Elixir の Flow [19] を用いたプログラミングスタイルが GPGPU のコードに容易に変換できるという仮説に基づいている．また GPU を直接操作するネイティブコードを記述するために，Elixir から Rust [7] で記述したネイティブコードを呼出すためのライブラリである Rustler [8] と，Rust で OpenCL [13] を用いて GPU を駆動する ocl [16] を採用して実装した．

本稿の以下の構成は次のとおりである: 第 2 章では Elixir コードから GPU を駆動するコードに変換する中核となるアイデアと Hastega の実装について提案する．第 3 章では我々が行なった実験の実行環境と，ベンチマークについての記述，結果と考察を示す．第 4 章ではインストールと設定のプロセスについて示し．設定容易性について議論する．第 5 章では本稿をまとめて将来課題について述べる．

```
1..1000
  |> Flow.from_enumerable()
  |> Flow.map(foo)
  |> Flow.map(bar)
  |> Enum.to_list
```

図 1 Flow を使ってリストを操作する Elixir コード
Fig. 1 Elixir code of list manipulation using Flow.

```
__kernel void calc(
  __global long* input,
  __global long* output) {
  size_t i = get_global_id(0);
  long temp = input[i];
  temp = foo(temp);
  temp = bar(temp);
  output[i] = temp;
}
```

図 2 図 1 の OpenCL コード
Fig. 2 OpenCL code of figure 1

2. Hastega の方針と実装

MapReduce モデル [3] に基づく Flow [19] を用いた並列プログラミングスタイル (**図 1**) は，GPU に採用されている SIMD(単一命令複数データ) アーキテクチャにそのまま適合する．すなわち，パイプライン演算子で繋がれている関数群 `foo |> bar` は「単一命令」に，範囲 `1..1000` によって生成されるリストは「複数データ」に，それぞれ適合する．このことから，Flow で書かれた Elixir のコードは GPU で実行できるコードに容易に変換できる．

図 1 を元に OpenCL で実行できるコードを**図 2** に示す．Flow のコードの命令列 `foo |> bar` をそのまま関数 `foo` と `bar` を続けざまに呼び出すコードに変換し，範囲 `1..1000` によって生成されるリストをあらかじめ配列にして `input` として与え，結果を `output` で受取るようにプログラムを構成する．

図 3 に Hastega のアーキテクチャを示す．我々は Rust [7] でネイティブコードを記述し，Rustler [8] と ocl [16] を用いた．このことによりとても設定が容易になる．プログラミング言語と OpenCL [13] と Hastega をそれぞれ 1 行程度のコマンドでインストールするだけである．これは，Python [20] や CUDA [11] と，CuPy [14] のような関連ライブラリと比べて大きな優位性を持っている．

[1] 北九州市立大学
University of Kitakyushu
[2] 有限会社デライトシステムズ
Delight Systems Co., Ltd.
[3] 京都大学
Kyoto University
[a)] zacky@kitakyu-u.ac.jp
[b)] mori@delightsystems.com
[c)] delightadmin@delightsystems.com
[d)] takase@i.kyoto-u.ac.jp

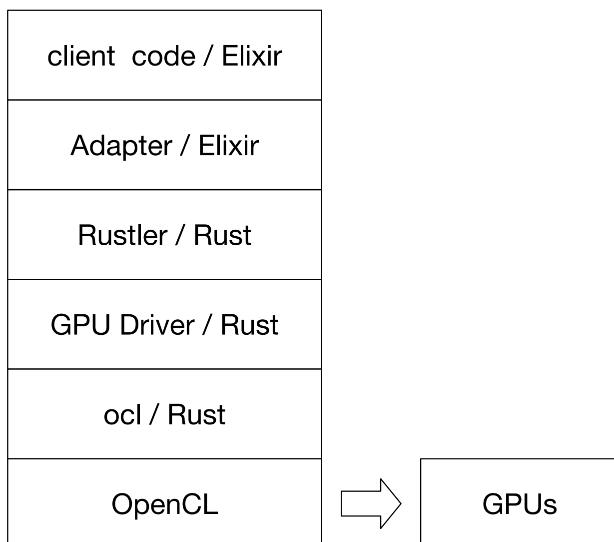

図 3 Hastega アーキテクチャ
Fig. 3 The Hastega Architecture

我々の実装は GitHub に公開されている[*1]．

3. パフォーマンス評価

3.1 評価環境

我々は素体によるロジスティック写像 [9] をベンチマークとして採用した．これは下記のような漸化式で表される．

$$X_{i+1} = \mu_p X_i(X_i + 1) \bmod p$$

これを採用した理由は，GPU のアセンブリコードに容易に変換できる整数演算を用いて負荷の高い計算をすることができるためである．

我々は Mac Pro (Mid 2010) と Google Compute Engine(GCE) [5] という 2 つの環境で評価した．

1. Mac Pro (Mid 2010) は 1 つの 2.8GHz のクアッドコア Intel Xeon の CPU と，16GB のメモリ，そして 1024MB のメモリを持つ ATI Radeon HD 5770 で構成される．
2. GCE の設定は次のとおりである．
   - Machine type: custom (8 vCPUs，16GB メモリ)
   - CPU platform: Intel Broadwell
   - GPU: NVIDIA Tesla K80 (x1)
   - Zone: us-west1-b

表 1 に用いたソフトウェアのバージョンを示す．

3.2 ベンチマーク

本実験では下記のベンチマークを用いた:

- Elixir_recursive: Elixir [18] のみで書かれている．ロジスティック写像の計算は 10 回の再帰呼出しで実装されている．
- Elixir_inlining: Elixir のみで書かれている．ロジスティック写像の計算は Flow の中にインライン展開されている．
- Rustler_CPU: Elixir と Rust [7] で書かれており，Rustler [8] を用いている．ロジスティック写像は Rust で書かれたネイティブコードで CPU で計算している．scoped-pool [15] によるスレッドプールを用いた非同期 NIF(Native Implemented Function) 呼出しを用いており，単一スレッドで動作する．
- Rustler_CPU_multi: Rustler_CPU と同様であるが，ロジスティック写像を rayon [17] による複数スレッドで計算する点が異なる．ただし，実装上の制約で rayon のスレッドプールは有効にできなかった．
- Rustler_GPU: Elixir と Rust で書かれており，Rustler を用いている．ロジスティック写像は Rust と ocl [16] を用いて書いたネイティブコードで OpenCL [13] 経由で GPU で計算されている．
- Empty: 実行効率の観点で比較するためのダミーベンチマークである．Elixir のリストと Rust のベクターの返還を含むが，ロジスティック写像の計算は含まない．
- Rust_CPU: Rust のみで書かれている．ロジスティック写像は単一スレッドの CPU で計算されている．
- Rust_CPU_multi: Rust_CPU と同様であるが，ロジスティック写像は rayon [17] を用いて複数スレッドで実行されている．かつ，rayon のスレッドプールが有効である．
- Rust_GPU: Rust だけで書かれている．ロジスティック写像は ocl を用いて OpenCL 経由で GPU で計算されている．
- Python_CPU: Python [20] と NumPy [12] で書かれている．
- Python_GPU: Python [20] と CuPy [14] で書かれており，GPU で実行する．

3.3 評価の結果と考察

表 2 にベンチマークの結果を示す．

- Elixir のみ (Elixir_recursive と Elixir_inlining) と比べて，Rustler_GPU は 4.43–8.23 倍，Rustler_CPU_multi は 5.68–6.97 倍高速である．
- Rustler_CPU_multi と Rustler_GPU は，Mac Pro (Mid 2010) と GCE で逆になっている．この理由は，rayon のスレッドプールがないことで，Linux ではパフォーマンスが悪化するためである．
- Rustler_GPU と Empty の差の比は 22.2–27.6 ％である．これが Elixir のリストと Rust のベクターを変換するオーバーヘッド等を除いた実質的な実行時間であ

[*1] LogisticMap: Benchmark of Logistic Map using integer calculation and Flow, available at https://github.com/zeam-vm/logistic_map

表 1 用いたソフトウェアのバージョン
Table 1 Versions of Software

	Mac Pro (Mid 2010)	GCE
OS	Sierra 10.12.6	ubuntu 16.04
Elixir [18]	1.6.6 (OTP 21)	1.6.6 (OTP 21)
Flow [19]	0.14	0.14
Rust [7]	1.27.0	1.27.0
OpenCL [13]	1.2	1.2
Rustler [8]	0.17.1	0.17.1
ocl [16]	0.18	0.18
rayon [17]	1.0	1.0
scoped-pool [15]	1.0.0	1.0.0
Python [20]	3.6.0 (Anaconda 4.3.0)	3.5.2
CUDA [11]	N/A	9.0 (in case of using CuPy), 9.2 (other)
NumPy [12]	1.11.3	1.14.3
CuPy [14]	N/A	4.1.0

ると考えられる．

- Python_CPU と比べて，Elixir のみは 1.17–1.68 倍，Rustler_GPU は 7.43–9.64 倍高速である．
- Python_GPU と比べて，Rustler_GPU は 3.67 倍高速である．
- Rustler_CPU と Rust_CPU の差の比は 62.0–70.0 %，Rustler_GPU と Rust_GPU の差の比は 32.2–35.2 %である．これは Ealang VM のオーバーヘッドであると考えられる．
- Rustler_GPU と比べて，Rust_GPU は 1.48–1.54 倍高速である．これが潜在的な最適化の余地であると考えられる．

4. 設定容易性の議論

表 3 に GCE [5] 上でのインストール・設定プロセスの比較を示す．

Hastega のインストールはビルドツールが自動で設定してくれるので煩雑ではない．必要なことは，OpenCL [13]/Elixir [18]/Rust [7]/Hastega をインストールすることだけである．

CuPy [14] における CUDA [11] もしくは OpenCL [13] のインストールは，Hastega と比べてより多くの作業手順を必要とする．その理由は，Cupy が古いバージョンの CUDA [11] を必要とするからである．この解決方法を知るために Stack Overflow [1] のような Q&A サイトを調べ上げる必要があった．

CuPy ではプログラミング言語は Python がプリインストールされているのに対し，Hastega では Elixir [18] と Rust [7] という 2 つのプログラミング言語のインストールと設定を必要とする点が提案手法の方が多く作業手順を要している．しかし，特別な設定は要しない．

結果として，設定プロセス全体では CuPy では設定について調査を要する分，設定に多大な時間がかかった．作業項目数は変わらないが，総じて Hastega の方が設定は容易である．

5. まとめと将来課題

Python とそのライブラリのパフォーマンスと設定の問題を解決するため，Flow を使った Elixir のコードを GPU の実行コードに変換することを提案した．これは，Elixir のコードは GPU で採用されている SIMD アーキテクチャに適合することに着眼した．

次にロジスティック写像のベンチマークで Hastega の効果を示した．Rust で記述したネイティブコードを Rustler と ocl を使って実装した．我々の実装は GitHub [21] に公開している．

Elixir と Rustler を用いた GPGPU の試行的な実装のパフォーマンスを評価して下記のような結果が得られた．

- Elixir と Ruslter を用いて GPU を駆動するコード (Hastega) は，CPU のみで実行する Elixir のみと比べて 4.43–8.23 倍，CPU のみで実行する Python と比べて 7.43–9.64 倍高速である．
- Hastega を GPU を駆動するネイティブコードと比較した時のパフォーマンスの違いは 1.48–1.54 倍である．
- さらに，Hastega は，GPU を駆動する Python コードと比べて 3.67 倍高速になった．

この実験を通じて，Erlang VM は，主なオーバーヘッドの原因であるリストとベクターの変換を削減するような最適化をするにはパフォーマンスが不十分であることがわかった．したがって，Hastega を，GPU を駆動してリストとベクターの変換を削減する最適化をするのに十分な能力を持つ新しい Elixir の処理系として実装する．

表 2 ベンチマーク結果
Table 2 The result of the benchmarks

			Mac Pro (Mid 2010) 2.8GHz Quad-Core Intel Xeon ATI Radeon HD 5770 (秒)	GCE Intel Broadwell vCPU:8 NVIDIA Tesla K80 (秒)
Elixir_recursive	Elixir	再帰呼出し	12.177	9.674
Elixir_inlining	Elixir	インライン展開	10.579	8.075
Rustler_CPU	Elixir / Rustler	CPU (単一スレッド)	7.691	6.098
Rustler_CPU_multi	Elixir / Rustler	CPU (マルチスレッド)	1.748	1.422
Rustler_GPU	Elixir / Rustler	Hastega: OpenCL (GPU)	2.388	1.176
Empty	Elixir / Rustler	ダミーベンチマーク	1.859	0.852
Rust_CPU	Rust	CPU	2.926	1.829
Rust_CPU_multi	Rust	CPU	0.669	0.374
Rust_GPU	Rust	OpenCL (GPU)	1.546	0.797
Python_CPU	Python	NumPy (CPU)	17.749	11.341
Python_GPU	Python	CuPy (GPU)	N/A	4.316

表 3 GCE 上でのインストール・設定プロセスの手順の比較
Table 3 Comparison of Steps of Installation and Setting Processes in GCE

	CuPy	Hastega
CUDA もしくは OpenCL のインストール	4 †	1
プログラミング言語のインストール	0	4
ライブラリのインストール	2	1

† 相当時間の StackOverflow による技術調査を要した

さらに設定容易性の評価を行った．その結果，CuPy と比べて，CUDA もしくは OpenCL のインストールの作業手順が大幅に簡素化されることがわかった．

我々には Elixir で数学と機械学習のライブラリを実装する計画がある．もちろん，これらにも Hastega を適用し，Python とその機械学習系・GPU 実装と性能を比較したい．

謝辞 整数演算のベンチマークで適切なものがないか相談したところ素体のロジスティック写像 [9] をご教示くださった上原聡先生，宮崎武氏に深く感謝する．

参考文献

[1] Atwood, J. and Spolsky, J.: Stack Overflow: Learn, Share, Build (2008). https://stackoverflow.com.

[2] Chollet, F.: Keras: The Python Deep Learning Library (2017). https://keras.io.

[3] Dean, J. and Ghemawat, S.: MapReduce: Simplified Data Processing on Large Clusters, *Commun. ACM*, Vol. 51, No. 1, pp. 107–113 (online), DOI: 10.1145/1327452.1327492 (2008).

[4] Fedrecheski, G., Costa, L. C. P. and Zuffo, M. K.: Elixir programming language evaluation for IoT, *2016 IEEE International Symposium on Consumer Electronics (ISCE)*, pp. 105–106 (online), DOI: 10.1109/ISCE.2016.7797392 (2016).

[5] Google: Compute Engine: Scalable, High-Performance Virtual Machines (2012). https://cloud.google.com/compute/.

[6] Google: TensorFlow: An Open Source Machine Learning Framework for Everyone (2017). https://www.tensorflow.org.

[7] Hoare, G.: Rust: Rust is a systems programming language that runs blazingly fast, prevents segfaults, and guarantees thread safety. (2006). https://www.rust-lang.org/.

[8] Josephsen, H. E. B.: Rustler: Safe Rust bridge for creating Erlang NIF functions (2015). https://github.com/hansihe/rustler.

[9] Miyazaki, T., Araki, S., Uehara, S. and Nogami, Y.: A Study of an Automorphism on the Logistic Maps over Prime Fields, *Proc. of The 2014 International Symposium on Information Theory and its Applications (ISITA2014)*, pp. 727–731 (2014).

[10] Nickolls, J., Buck, I., Garland, M. and Skadron, K.: Scalable Parallel Programming with CUDA, *ACM SIGGRAPH 2008 Classes*, SIGGRAPH '08, New York, NY, USA, ACM, pp. 16:1–16:14 (online), DOI: 10.1145/1401132.1401152 (2008).

[11] NVIDIA: CUDA: Develop, Optimize and Deploy GPU-accelerated Apps (2006). https://developer.nvidia.com/cuda-toolkit.

[12] Oliphant, T.: NumPy: NumPy is the fundamental package for scientific computing with Python. (2005). http://www.numpy.org.

[13] OpenCL Working Group: OpenCL: The Open Standard for Parallel Programming of Heterogeneous Systems (2008). https://www.khronos.org/opencl/.

[14] Preferred Networks: CuPy: A NumPy-compatible matrix library accelerated by CUDA (2005). https://cupy.chainer.org.

[15] Reem, J.: scoped-pool: A flexible thread pool providing scoped and static threads. (2016). https://github.com/reem/rust-scoped-pool.

[16] Sanders, N.: ocl: OpenCL for Rust (2015). https://github.com/cogciprocate/ocl.

[17] Stone, J. and Matsakis, N.: Rayon: A data parallelism

library for Rust (2014). https://github.com/rayon-rs/rayon.

[18] Valim, J.: Elixir: Elixir is a dynamic, functional language designed for building scalable and maintainable applications. (2013). https://elixir-lang.org.

[19] Valim, J.: Flow: Computational parallel flows on top of GenStage (2017). https://github.com/elixir-lang/flow.

[20] van Rossum, G.: Python: Python is a programming language that lets you work quickly and integrate systems more effectively. (1991). https://www.python.org.

[21] Yamazaki, S., Mori, M., Ueno, Y. and TAKASE, H.: LogisticMap: Benchmark of Logistic Map using integer calculation and Flow (2018). https://github.com/zeam-vm/logistic_map.