Elixir におけるC言語コード生成・最適化の試み

山崎 進[1,a)]　久江 雄喜[1]

概要：我々は Pelemay Super-Parallelism と称する関数型言語 Elixir から C 言語へのコード生成・最適化を行う処理系を研究開発している．これは，Elixir には Elixir のコードを AST に変換する機構が備わることを利用し，並列化可能なコードを検出して C 言語コードを生成し，Auto Vectorization を利用して SIMD 並列化を行う処理系である．2019 年 10 月に公開したバージョン 0.0.4 では，あらかじめテンプレートとして定義された C 言語コードにコードを埋め込む方式のコード生成を実装し，元の Elixir コードと比べて約 2 倍の高速化が図れた．2019 年 11 月現在研究開発中のバージョン 0.1.0 系列では，コード生成の方式を大幅に見直し，柔軟なコード生成を図れるようにする予定である．本発表では，その中で最も特徴的な型検査の最適化方法について提案する．これは，動的型言語である Elixir の型の特性と，NIF という Elixir とネイティブコードの間の FFI を利用して，型検査と型推論を並行して実行することで，型検査の最適化を図る方式である．今後，実装と性能評価を行う．

キーワード：並列化, 型検査最適化, Elixir

1. はじめに

Elixir (エリクサー) [28] は 2012 年に José Valim が開発した並列プログラミング言語である．Elixir は関数型言語 Erlang [6] を母体としており，並列プログラミングのための数々の優れた特長を有する．

Elixir を用いることで，たとえばウェブシステムのレスポンス性能を大きく改善することができる．Fedrecheski ら [8] によると，Java では毎秒 1,200 リクエスト程度で急速にレスポンス性能が悪化してしまったのに対し，Elixir では毎秒 1,800 リクエスト程度まで耐えられる．

我々は Elixir のもつ並列プログラミングの特長をさらに活かすために，2018 年から Hastega(ヘイスガ) を研究開発した [14], [29], [30], [31], [32], [33]．ただし Hastega という名称はファイナルファンタジーに由来するもので，スクエアエニックスに著作権があると考えられるものであることから，2019 年 8 月に Pelemay(ペレメイ) に改称した [34]．現在では，より包括する概念として Pelemay ファミリー (**図 1**) として研究提案をしようとしており，従来の Hastega に対応するものは，Pelemay Super-Parallelism と呼んでいる．

Pelemay Super-Parallelism は，ウェブのサーバー・エッジ・クライアント上にあるマルチコア CPU や GPU に負荷分散しつつ SIMD 並列計算を行うコードを生成することを目指した処理系である．2019 年 11 月現在では CPU コアの 1 つを使った SIMD 計算のコードを生成する [34]．今後開発するバージョン 0.2.0 系列にてマルチコア CPU や GPU の駆動をすることを目指している．

本発表では，Pelemay Super-Parallelism のバー

[1] 北九州市立大学
[a)] zacky@kitakyu-u.ac.jp

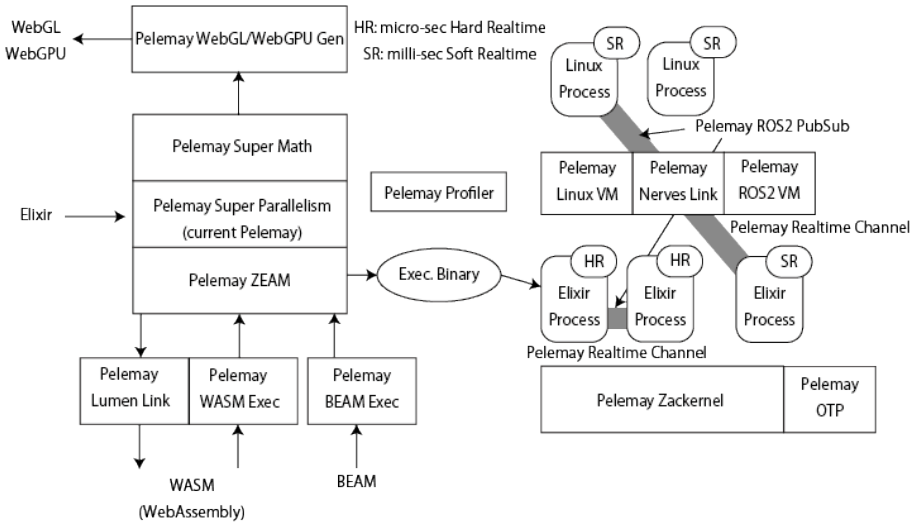

図 1: Pelemay ファミリ構想図

ジョン 0.0.4 で達成した性能向上について紹介するとともに，バージョン 0.1.0 系列で実装される予定のコード生成の方式の一部について提案・報告する．これは次のような特徴を持つ:

- Elixir に備わるメタプログラミング機構を利用してリストと 3 つ組からなる AST に変換し，我々が提唱する Elixir Zen Style [31], [33] から AST の変換・最適化を行う．
- Elixir の AST から C 言語に近い中間表現である ZEAM IR を生成し，Clang [17] と GCC [9] のそれぞれに対応する，Auto vectorization [10], [16] を行う C 言語コードを生成する．
- 動的型言語である Elixir の型の特性と，NIF[7] という Elixir とネイティブコードの間の FFI を利用して型検査と型推論を並行して実行することで，型検査の最適化を図る
- Elixir で記述された分岐構造を考慮して分岐予測の最適化を実施する．
- BigNum などネイティブコードでは処理できない値に遭遇した場合には非ネイティブコードで実行しなおす．

本発表では，このうち型検査の最適化について提案する．

我々の実装は GitHub に公開されている[*1]．

以降の本発表の構成を紹介する: 第 2 章では，並列化がこれまで以上に求められる背景について説明する．第 3 章では，プログラミング言語 Elixir とウェブアプリケーションフレームワーク Phoenix について紹介する．第 4 章では，我々が提唱･推進する Elixir Zen スタイルについてコード例と利点を紹介する．第 5 章では, Pelemay Super-Paralellism の着想と原理についてについて紹介し，本発表の焦点を提示する．第 6 章では，2019 年 10 月に公開した Pelemay Super-Paralellism バージョン 0.0.4 での処理の流れについて紹介する．第 7 章ではバージョン 0.0.4 の性能評価結果を示す．第 8 章では現在研究開発中のバージョン 0.1.0 系列で予定している処理の流れについて説明する．第 9 章では型検査と型推論を並行実行するコード最適化について提案する．最後に第 10 章で本発表をまとめる．

2. 並列化が求められる背景

Cisco [5] によると，インターネット全体の IP トラフィックは，2017 年に毎月 122 エクサバイト増加している．この増加率は年々増大しており，2022 年には毎月 396 エクサバイト増加すると見込んでいる．この増加のほとんどは動画によるものであるが，接続デバイス数で見ると M2M が急速

[*1] Pelemay, available at `https://github.com/zeam-vm/pelemay/`

に増加している．また，IDC [27] によると，インターネット上のデータの総量は 2013 年に 4.4 ゼタバイトであるが，2020 年には 44 ゼタバイトに増加すると見込まれている．これらが示すように世界に流通する情報量は急速に増大している．

また，第 5 世代移動通信システム (5G) により通信速度も飛躍的に向上する．NTT ドコモが 2019 年 9 月から提供する 5G プレサービスの最大通信速度はミリ波の受信時で 3.2Gbps に及ぶ [22]．実証実験の段階では，ニューヨークでの事例 [24]，シカゴでの事例 [26] で，500Gbps 以上の通信速度を達成したと言われている．また遅延に対する要求も厳しく，要求条件としては無線区間の遅延を 1ms 以下にすることが求められる [15]．

これらの問題に対処するためには，情報を処理する計算能力を情報量の流通量・流通速度と同等以上に高める必要がある．しかし，Hennesy と Patterson [12] によると，2003 年までは年々順調にクロック周波数が増加していたのに対し，2003 年ごろから頭打ちとなってしまっている．これはクロック周波数が増大すると消費電力と発熱量も増大するが，電源供給量と熱伝導率および冷却能力が追いつかなくなり，常温の環境下で安定して動作させることが不可能になってしまうためである．

2003 年以降のプロセッサの進化はクロック周波数の増加ではなくコア数の増加によっている．Intel が 2006 年に販売した Core シリーズの最上位プロセッサである Intel Core 2 Extreme X6800 は，クロック周波数が 2.93GHz，物理コア数は 2，論理コア数は 4 である．一方，2017 年に販売した Core シリーズの最上位プロセッサである Intel Core i9 7980XE は，クロック周波数が 2.6GHz，物理コア数は 18，論理コア数は 36 である．

この方向性をもっと極端に推し進めたのが，GPU や，KiloCore[3]，The Cerebras Wafer-Scale Engine (WSE) [4] である．現代的な GPU は SIMD 方式で設計されることが一般的であり，MIMD 方式と比べて単純化されることから，極めて大きいコア数が実現されている．たとえば NVIDIA TITAN RTX では，SIMD コアが 4000 以上にも及ぶ．KiloCore[3] は，MIMD 方式のコアで初めて 1000 以上を達成したプロセッサである．WSE[4] はウェハーサイズの巨大なプロセッサであり，AI 処理に最適化されたコアが 40 万にも及ぶ．

クロック数が順調に伸びていた時代ではソフトウェアを何も変えなくても順調に性能が向上していたであろう．一方，クロック周波数が伸びずにコア数が増加する状況下で性能を向上させるためには，並列プログラミングが求められる．そこで，近年次々と，さまざまな並列プログラミングモデルに基づいたプログラミング言語が提案されている．Elixir や Pelemay Super-Parallelism は，その中の 1 つに位置付けられる．

3. Elixir (エリクサー)

Elixir(エリクサー) [28] は 2012 年に José Valim が開発した並列プログラミング言語である．José は Ruby on Rails [11] のコミッタの 1 人でもあることから，Elixir は Ruby [18] の影響を強く受けている．Elixir 開発当時では，Ruby には並列プログラミング機構が十分備わっていなかったことから，アクターモデル [13] に基づく並行プログラミングモデルを採用するなど，並列プログラミングのための数々の優れた特長を有する関数型言語 Erlang [6] を母体とした．

このような背景から，Elixir は Ruby on Rails と同等のウェブアプリケーションフレームワークを実現するのに必要な機能を盛り込んだ言語仕様を備えており，その 1 つがメタプログラミング機構である．Elixir のメタプログラミング機構は，LISP [19] のメタプログラミング機構と似ており，タプルとリストで構成される抽象構文木 (AST) を操作することで行う．プログラムコードから AST を生成する関数が `quote` であり，AST に新しいコードや値を挿入する関数が `unquote` である．Elixir は LISP 同様，核となる言語仕様 (Elixir カーネル言語) をメタプログラミングを用いて拡張している．Elixir は，Elixir カーネル言語を，Erlang VM のバイトコード BEAM にコンパイルすることで，Erlang VM で実行できるようにしている．

以上のようなことから，「Elixir は，Erlang を母とし，Ruby を父とし，LISP を祖父に持つプログ

ラミング言語である」と俗に言われる．

Ruby における Ruby on Rails に相当する Elixir のウェブアプリケーションフレームワークが，Phoenix [20] である．Phoenix はレスポンス性能が極めて高いことから，オンラインゲームのバックエンドとして用いられる採用事例が目立つ．Elixir と Phoenix は後発であるにも関わらず，普及が目覚ましく，2016 年の時点で 1109 の企業がプロダクション採用している [1]．

Elixir とその母体の Erlang， Phoenix には，耐障害性が極めて高いという特徴がある．Elixir は Erlang 由来の並行プログラミング機構として，メモリ空間を GC を含めて完全に分離するプロセスモデルを採用している．また，監視プロセスを用いて，異常終了したプロセスを検知して再起動する仕組みを備えている．Phoenix ではこれらを利用して，高負荷をかけたり不正なデータを送信したりしても，異常終了したプロセスを監視プロセスによって再起動させる設計にしている．

なお，我々は [33] にて「Elixir と Phoenix の名は，ファイナルファンタジーに登場するアイテムと幻獣の名称に由来する」と紹介したが，José Valim 本人が我々宛の私信にてこれを否定した．それによると，Elixir は Erlang と同じ E で始まる単語で語感が良かったものから選んだようであるが，記憶が定かではないとのことである．ただ，Elixir は，もともとは錬金術師が求めた不老不死の薬の名称であり，一方 Phoenix は，自分の身を炎の中に投じて再び蘇る伝説上の不老不死の炎の鳥の名称であることから，Erlang の持つ耐障害性が不老不死のイメージにつながり，命名に至ったのではないか？と我々は推測する．

Elixir のコード例については，次の第 4 章で紹介する．

4. Elixir Zen Style

図 2 は，関数型言語で一般的な再帰呼出スタイルによる Elixir プログラム例である．コードの説明は次のとおりである:

- `1..1_000_000` は，1 から 1,000,000 までの整数の範囲を表す．
- `Enum.to_list` は，引数で与えられた値をリストに変換する．
- `|>` は**パイプライン演算子**である．パイプライン演算子は 2 項演算子であり，左辺の値を右辺の関数の第 1 引数として与え，右辺の関数の引数を第 2 引数以下に繰り下げて，関数を呼び出す．
- したがって，`1..1_000_000 |> Enum.to_list` は，`Enum.to_list(1..1_000_000)` と等価である．すなわち，1 から 1,000,000 までの整数を要素として持つリストを生成する．
- `1..1_000_000 |> Enum.to_list |> R.func()` は，パイプライン演算子記法で，`R.func(Enum.to_list(1..1_000_000))` と等価である．パイプライン演算子により，コードを読むときに，左から右へ，上から下へ，自然な流れで読むことができるようになり，括弧のネスティングにより可読性が落ちることを防ぐ．
- `R.func` は，モジュール `R` に定義されている関数 `func` のことである．
- 13〜16 行目の `defmodule M do ... end` は，`...` を本体として持つモジュール `M` を定義する．
  - `def func(arg) do ... end` により，`...` を本体として持つ，引数 `arg` の関数 `func` を定義する．`do ... end` が 1 行で記述できる場合には，`def func(arg), do: ...` と書いても良い．
  - 引数の数をアリティと呼ぶ．異なるアリティを持つ同名の関数は，異なる関数として区別される．Elixir では関数を厳密に区別するために，たとえば `R.func/1` というような記法を用いる．これは，モジュール `R` で関数名 `func`，アリティが 1 の関数，という意味である．
  - 関数 `M.foo/1` は，第 1 引数の値を 2 倍した値を返す関数である．同様に関数 `M.bar/1` は，第 1 引数の値を 1 加えた値を返す関数である．このような関数は Lisp 同様に無名関数

```elixir
1..1_000_000
|> Enum.to_list
|> R.func()

defmodule R do
    def func( [] ), do: []
    def func( [ head | tail ] ) do
        [ head |> M.foo() |> M.bar()
        | func(tail) ]
    end
end

defmodule M
    def foo(n), do: n * 2
    def bar(n), do: n + 1
end
```

図 2: 再帰呼出スタイルの Elixir コード例

Fig. 2　An Elixir code example using recursive call

で定義することもできるが，本報告では説明を割愛した．

- 同じアリティを持つ同名の関数の定義が複数あるときには，なんらかの条件で分岐して実行される．たとえば，関数 `R.func/1` では，空リスト `[]` を引数にする場合が先に定義されていて，リスト `[head | tail]` が引数である場合が後で定義されている．この場合，先に引数が空リストにマッチするかを評価し，マッチすれば空リストの場合の関数定義を実行する．マッチしなければ，次の定義であるリストにマッチするかを評価し，マッチすればリストの場合の関数定義を実行する．マッチしなければ，`FunctionClauseError` を発生させる．このように動的に呼び分ける仕組みを，**関数パターンマッチ**と呼ぶ．

```java
int i;
int [] array = new int[1000000];
for(i = 0; i < 1000000; i++)
    array[i] = i + 1;
for(i = 0; i < 1000000; i++)
    array[i] = foo(array[i]);
for(i = 0; i < 1000000; i++)
    array[i] = bar(array[i]);
```

図 4: ループスタイルの Java コード例

Fig. 4　A Java code example using loop

```elixir
1..1_000_000
|> Enum.map(&M.foo(&1))
|> Enum.map(&M.bar(&1))

defmodule M
    def foo(n), do: n * 2
    def bar(n), do: n + 1
end
```

図 3: Enum.map スタイルの Elixir コード例 (Elixir Zen Style)

Fig. 3　An Elixir code example using Enum.map

- 関数 `R.func/1` の引数が空リスト `[]` である場合には，空リストを返す．
- 関数 `R.func/1` の引数が空リストではないリストである場合には，LISP の `car` にあたる部分を `head` に，`cdr` にあたる部分を `tail` にそれぞれ束縛する．返す値は，`head` に `M.foo/1` と `M.bar/1` を適用して得られた値を先頭要素とし，`R.func/1` に `tail` を与えて再帰的に呼び出して得られた値を末尾要素としたものを返す．
- これらにより全体として，1 から 1,000,000 までの要素からなるリストの各要素に対し，関数 `M.foo/1` と `M.bar/1` を適用して得られる値からなるリストを生成する．

これと同じ結果が得られる別の書き方のコードを**図 3** に示す．このコードの説明は次のとおりである:

- `Enum.map/2` は，第 1 引数をリストとして扱い，その各要素に対し，第 2 引数で与えられる関数呼出を行う．
- `&M.foo(&1)` は，無名関数定義の簡易記法で表しており，意味としては`&1` は第 1 引数を表しており，`M.foo/1` にそのまま第 1 引数を渡して呼び出しをする．
- `Enum.map/2` で抽出された各要素が第 1 引数

```
__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 5: 図 3 のコードを OpenCL のコードに手で変換したコード

Fig. 5 A sample code of OpenCL transformed by hand from the code of Fig. 3

として与えられるので，この値を当てはめて `M.foo/1` を呼び出すことになる．

- したがってこれらにより，`1..1_000_000 |> Enum.map(&M.foo(&1)) |> Enum.map(&M.bar(&1))` により，1 から 1,000,000 までの各要素に対し `M.foo` と `M.bar` を適用した値を要素とするリストを返す．

参考までにループを使って書いた場合を**図 4** に示す．Elixir ではループを使っては記述できないので，Java で書いた．

我々は Lonestar ElixirConf 2019[31] にて，図 3 のスタイルが最もシンプルで，読みやすく保守性に優れることを指摘し，このスタイルを **Elixir Zen Style** と呼ぶことにしよう，と提案した [31], [33].

5. Pelemay Super-Parallelism

Pelemay Super-Parallelism (旧名称 Hastega) [14], [29], [30], [31], [32], [33], [34] は，Elixir Zen Style (図 3) のコードを最適化してネイティブコード化するプログラミング言語処理系である．

図 3 のコード相当のプログラムを OpenCL [23] で記述したコードを **図 5** に示す．Pelemay Super-Paralellism の原理は，図 3 と 図 5 のコードに類似性が多く，機械的な変換で Elixir Zen Style のコードから OpenCL のコードにコンパイルできるだろう，という着想に由来する [29].

このようなコードは SIMD (Single Instruction Multiple Data) アーキテクチャで並列実行するのに向いている．最近の Intel CPU には SIMD 命令が備わっており， GPU も SIMD アーキテクチャに基づいていることから，CPU や GPU で高速実行できる可能性が高い．

しかも Elixir Zen Style のコードは容易に並列性を稼いだり調整したりすることができる．図 3 のコードは，最大 1,000,000 並列で実行できることが明白である．`Enum.map` の中に記述されている処理は，互いに依存関係や副作用がないことから，並列に実行してもかまわないことが読み取れる．このような依存関係や副作用の解析は，`Enum.map` の中も Elixir Zen Style で記述する限りでは容易であると考えられる．しかも，1,000,000 並列の処理を実際の SIMD コアにどのように割り当てるかについては，とくにそのことを指定したり依存したりするような記述がないことから，処理系に自由な裁量が与えられていると考えて良い．

新井ら [2] は，並列化には，並列実行によって性能向上に貢献するようなコードブロックを発見することが重要であるとしている．そこで，Pelemay Super-Parallelism では，次の 2 つの要素技術で構成する:

- Elixir Zen Style のような，並列実行によって性能向上に貢献するようなコードブロックをソースコード中から発見する
- 発見したコードブロックを並列化するコードを生成・最適化する

本発表は後者について論じる．前者については，[33] である程度論じたほか，現在，さらなる研究成果をまとめているところである．

図 6 のコードは Pelemay Super-Parallelism のコード例である．

- 2〜3 行目は Pelemay をライブラリとして読み込む設定である．
- 5〜10 行目の `defpelemay` は，後続する `do ... end` の範囲のプログラムコードを Pelemay 処理系に渡し，コード最適化する．Pelemay Super-Parallelism は，この区間のコードブロックに関し，ホットスポットであるかどうかに関わらず，並列化可能な部分は全て並列化する方針をとっている．

```
1  defmodule P
2    require Pelemay
3    import Pelemay
4
5    defpelemay do
6      def func list do
7        list
8        |> Enum.map(& &1 + 1)
9      end
10   end
11 end
```

図 6: Pelemay コード例
Fig. 6　An Example Code of Pelemay

- 8 行目の `& &1 + 1` は第 1 引数で与えられた値に 1 加える無名関数である．`|> Enum.map(& &1 + 1)` によって，パイプラインで渡されたリストの各要素に 1 加える．

Pelemay 処理系 (バージョン 0.0.4) は**図 7** のような C 言語のコードを生成する．これは Elixir/Erlang の FFI である Native Implemented Function (NIF) [7] のコードであり， Clang [17] の Auto Vectorization [16] を用いて SIMD 命令を生成している．

- `ERL_NIF_TERM` は Elixir/Erlang の任意の項 (term) を表す構造体である．
- `ErlNifEnv *env` は実行時環境である．
- `argc` は NIF に与えられた引数の個数を表し，`argv` は引数の配列を表す．
- `__builtin_expect(exp, bool)` は Clang で用意されている分岐予測のヒントを与える組込みマクロで，式 `exp` の値が真偽値 `bool` で与えられる値である可能性が高いとして，分岐予測のコードを生成する．
- `enif_make_badarg` により引数が合っていないという例外を発生させる．4〜6 行目により引数の個数が 1 でなかった場合は例外を発生して終了する．
- `vec_long`, `vec_double` によって引数で与えられたリストを配列として受け取る．`vec_l` はその個数である．
- `enif_get_long_vec_from_list` と `enif_get_double_vec_from_list` は Pelemay で用意されている，リストを配列として受け取る関数で，それぞれ 64 ビット整数，浮動小数点数として受け取る．リストの各要素が整数である場合と浮動小数点数である場合の両方に対応させる．
- `#pragma clang loop vectorize_width(loop_vectorize_width)` によって Clang に SIMD 命令を含む auto vectorization コードを生成させる指示を出している．ここではベクターの幅を指定している．これにより，後続するループについて，指定されたベクター幅で SIMD 命令を用いてコード生成される．
- `enif_make_list_from_double_vec` と `enif_make_list_from_long_vec` は，Pelemay で用意されている，配列をリストに変換して返す関数で，それぞれ浮動小数点数，64 ビット整数の配列から変換する．

6. バージョン 0.0.4 の処理の流れ

Pelemay Super-Parallelism バージョン 0.0.4 は 2019 年 10 月末にリリースされた．これは次のような流れでコンパイルを行う:

1. Elixir に備わっているメタプログラミング機構により，`defpelemay do ... end` の区間を認識し，Pelemay コンパイラに処理を渡す [33].
2. 区間内の関数と，その中に記述されている `Enum.map` を含む記述，`Enum.map` の中の匿名関数を認識する [33].
3. NIF のコードのスタブを生成する．
4. 図 7 のような用意されているテンプレートをもとにネイティブコードを生成する．このテンプレートでは，Auto-Vectorization と分岐予測に関するヒントを含むコードとなっている．
5. C 言語コードを Clang に渡してコンパイルする．

7. バージョン 0.0.4 の性能評価

バージョン 0.0.4 について性能評価を行った．**表 1** に全ての評価環境に共通する特性を，**表 2** に

```
1  static ERL_NIF_TERM
2  map_plus(ErlNifEnv *env, int argc, const ERL_NIF_TERM argv[])
3  {
4    if (__builtin_expect((argc != 1), false)) {
5      return enif_make_badarg(env);
6    }
7    long *vec_long;
8    size_t vec_l;
9    double *vec_double;
10   if (__builtin_expect((
11     enif_get_long_vec_from_list(env, argv[0], &vec_long, &vec_l)
12     == fail), false)) {
13     if (__builtin_expect((
14     enif_get_double_vec_from_list(env, argv[0], &vec_double, &vec_l)
15     == fail), false)) {
16       return enif_make_badarg(env);
17     }
18 #pragma clang loop vectorize_width(loop_vectorize_width)
19     for(size_t i = 0; i < vec_l; i++) {
20       vec_double[i] = ((vec_double[i])+1); //  この部分でdouble時のコードを生成する
21     }
22     return enif_make_list_from_double_vec(env, vec_double, vec_l);
23   }
24 #pragma clang loop vectorize_width(loop_vectorize_width)
25   for(size_t i = 0; i < vec_l; i++) {
26     vec_long[i] = ((vec_long[i])+1); //  この部分でinteger時のコードを生成する
27   }
28   return enif_make_list_from_long_vec(env, vec_long, vec_l);
29 }
```

図 7: Pelemay によるコンパイル結果 (バージョン 0.0.4)
Fig. 7　The result compiled by Pelemay 0.0.4

評価環境ごとに異なる特性を示す．Elixir のベンチマークツールである Benchfella [25] を用いた．

ベンチマークプログラムとしては，素体によるロジスティック写像 [21] を採用した．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，コンパイラで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．

4096 個の要素からなるリストを生成し，Enum をそのまま用いた場合と Pelemay Super-Parallelism バージョン 0.0.4 を用いた場合について，リストの各要素に対し 10 回ずつ素体によるロジスティック写像を計算するコードの実行 1 回にかかる平均実行時間を計測した．

表 3 に実験結果を示す．Enum をそのまま実行する場合に比べて，Pelemay 0.0.4 で実行した場合，iMac Pro で 2.39 倍，Ryzen で 1.98 倍の速度向上になる．

8. バージョン 0.1.0 系列の処理の流れ

バージョン 0.1.0 系列は，2019 年 11 月現在研究開発中のリリース候補である．ネイティブコード生成について大幅に改良を加える予定である．

現時点では次のような流れでコンパイルを行う予定である:

1. Elixir に備わっているメタプログラミング機構により，`defpelemay do ... end` の区間を認識し，Pelemay コンパイラに処理を渡す [33].
2. 区間内の関数と，その中に記述されている `Enum.map` を含む記述，`Enum.map` の中の匿名関数を認識する [33].

3. NIF のコードのスタブを生成する．その際，型エラー回復のためのコードを生成する．
4. ZEAM IR を生成し，それに基づいてネイティブコードを生成する．その際，型検査を最適化し，Auto-Vectorization と分岐予測に関するヒントを含むコードを生成する．
5. C 言語コードを Clang もしくは GCC に渡してコンパイルする．

9. 型検査と型推論を並行実行するコード最適化

NIF[7] では，引数は ERL_NIF_TERM 型で与えられる．引数がリストだった場合には，空リストであるか，リストの各要素は ERL_NIF_TERM 型で与えられる．引数が範囲 (例えば 1..10) だった場合には，マップとして表されており，アトム:__struct__ をキーとした値 Elixir.Range を持ち (すなわち Range 型の構造体である)，アトム:first，:last をキーとした時の値が，それぞれ始点，終点であるように与えられる．

ERL_NIF_TERM 型の変数は NIF で提供される各種関数で C 言語の型の変数に変換するが，この際に変換先の型に変換できるか判定する型検査を伴う．NIF はそれとは別に型検査を行う関数を提供している．数値の場合は，変数が数であるかを判定する型検査関数 enif_is_number() の他に，16 ビット整数，32 ビット整数，64 ビット整数のそれぞれ符号ありと無し，倍精度浮動小数点数との変換関数が備わっており，それぞれの値に変換できるかどうかを判定する型検査を伴っている．

数の型検査の結果について，**表 4** にまとめた．特に注目してほしいのが，64 ビット整数の範囲を超える場合 (〜LLONG_MIN - 1，ULLONG_MAX〜) である．これらの場合は，enif_is_number によって数であるとは判定されるものの，NIF 関数を用いて値を取得することができない．この場合は，NIF では扱えないので，適切に処理する必要がある．

バージョン 0.1.0 系列では，型検査と型推論を並行実行することで，型検査に関するコード最適化を行う方法を行う．アイデアとしては次のとおりである:

- 与えられた引数それぞれについて型検査を行う．
- 引数が空リストであった場合は，空リストとして扱う．
- 引数が範囲型であった場合には，始点から終点までの整数を要素として持つ単一要素型の整数リスト (後述) として扱う．
- 引数がリストであった場合には，第 1 要素から順番に型検査をしていく．整数型だった場合には値の範囲や型の頻度も記録していく．
  - もし全ての要素が単一の型である場合 (**単一要素型**と称する) には，以後，このリストの要素に関する型検査は省略できる．
    - さらに要素の型が数だった場合，浮動小数点型であれば，double 型の配列に変換する．エラーになった場合は整数型であると判定する．整数値の範囲が int 型，unsinged int 型，long 型，unsigned long 型，ErlNifInt64 型 (Erlang が用意している符号付 64 ビット整数型)，ErlNifUInt64 型 (Erlang が用意している符号なし 64 ビット整数型) の順で型検査して値の範囲が収まるならば，それぞれの型の配列に変換できる．
    - 整数値の範囲が収まらない場合は NIF で扱うことができないので，型エラーとする．この後，Elixir 側でエラー回復するためのコードを実行する．
    - 要素の型が数ではない場合は，当てはまる型を記録した上で，ERL_NIF_TERM のまま記録する．以後はこの記録した型であるものとして，必要に応じて値を取り出す．
  - もしほとんどの要素の型が同一の型で，時々異なる型が混じるというような場合 (**一部複合要素型**と称する) は，配列に対応するビットベクタを用意し，ビット値が 0 の時には多くの場合に当てはまる方の型であるとして型検査を省略し，ビット値が 1 の時のみ ERL_NIF_TERM のまま記録して型検査

を行って値を取り出す，というようにする．ERL_NIF_TERM を用いる場合は，別途，当てはまった型を記録する．

– これらの場合に該当せず，異なる型が完全に混在する場合 (**複合要素型**と称する) には，ERL_NIF_TERM の配列として保持し，値の取り出しの際に型検査を行うことにする．ERL_NIF_TERM を用いる場合は，別途，当てはまった型を記録する．

- 引数がそれ以外の型であった場合には，対応する型の変数で取り出す．

10. まとめと将来課題

本発表では，Elixir の並列処理性能を向上させる Pelemay Super-Paralellism について，2019 年 10 月に公開したバージョン 0.0.4 の性能評価と，2019 年 11 月現在研究開発中のバージョン 0.1.0 系列の型検査の最適化について説明した．

将来課題としては，バージョン 0.1.0 系列を実装し性能評価することが挙げられる．これが実装できた段階で，バージョン 0.2.0 系列でマルチプロセッサ，マルチコア，GPU を駆動できるようにする予定である．

謝辞　本研究の一部は，北九州産業学術推進機構 (FAIS) 新成長戦略推進研究開発事業「シーズ創出・実用性検証事業」，および中小企業庁戦略的基盤技術高度化支援事業の支援を受けた．Plataformatec の José Valim，東京工業大学の増原英彦先生，カラビナテクノロジーの森正和氏，京都大学の高瀬英希先生，東海大学の大川猛先生にはとくに有益な助言を多数いただいた．

参考文献

[1] Adams, J.: Elixir Users' Survey 2016 Results (2016). https://www.dailydrip.com/blog/elixir-users-survey-2016-results.html.

[2] 新井淳也，前田俊行，石川　裕，米澤明憲：Glasgow Haskell Compiler を用いたプロファイル駆動型自動並列化機構の簡潔実装，情報処理学会論文誌プログラミング（PRO），Vol. 5, No. 1, pp. 37–37 (2012).

[3] Bohnenstiehl, B., Stillmaker, A., Pimentel, J. J., Andreas, T., Liu, B., Tran, A. T., Adeagbo, E. and Baas, B. M.: KiloCore: A 32-nm 1000-Processor Computational Array, *IEEE Journal of Solid-State Circuits*, Vol. 52, No. 4, pp. 891–902 (online), DOI: 10.1109/JSSC.2016.2638459 (2017).

[4] Cerebras Systems: The Cerebras Wafer-Scale Engine (2019). https://www.cerebras.net.

[5] Cisco: Cisco Visual Networking Index: Forecast and Trends, 2017–2022 (2018). White Paper.

[6] Ericsson: Erlang Programming Language (1998). https://www.erlang.org.

[7] Ericsson AB.: NIFs: Erlang Interoperability Tutorial (2000). http://erlang.org/doc/tutorial/nif.html.

[8] 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).

[9] Free Software Foundation, Inc.: GCC, the GNU Compiler Collection (1985). https://gcc.gnu.org.

[10] Free Software Foundation, Inc.: Auto-vectorization in GCC (2008). https://gcc.gnu.org/projects/tree-ssa/vectorization.html.

[11] Hansson, D. H.: Ruby on Rails: A web-application framework that includes everything needed to create database-backed web applications according to the Model-View-Controller (MVC) pattern (2004). https://rubyonrails.org.

[12] Hennessy, J. L. and Patterson, D. A.: *Computer Architecture: A Quantitative Approach*, Morgan Kaufmann, 6th edition (2017).

[13] Hewitt, C., Bishop, P. and Steiger, R.: A Universal Modular ACTOR Formalism for Artificial Intelligence, *Proceedings of the 3rd International Joint Conference on Artificial Intelligence*, IJCAI'73, San Francisco, CA, USA, Morgan Kaufmann Publishers Inc., pp. 235–245 (1973).

[14] 久江雄喜，山崎　進：Hastega: Elixir プログラミングにおける線形回帰の SIMD 命令による並列化，第 122 回情報処理学会プログラミング研究会, 福山, 広島，情報処理学会プログラミング研究会 (PRO)，Vol. 2018, No. 4, 東京，p. (5) (2019).

[15] 岸山祥久，ベンジャブールアナス，永田　聡，奥村幸彦，中村武宏：ドコモの 5G に向けた取組み—2020 年での 5G サービス実現に向けて—，Vol. 23, No. 4, pp. 6–17 (2016).

[16] LLVM Project: Auto-Vectorization in LLVM (2003). https://llvm.org/docs/Vectorizers.html.

[17] LLVM Project: Clang: a C language family frontend for LLVM (2003). https://clang.llvm.org.

[18] まつもとゆきひろ: Ruby Programming Language (1995). https://www.ruby-lang.org/en/.

[19] McCarthy, J.: Recursive Functions of Symbolic Expressions and Their Computation by Machine, Part I, *Commun. ACM*, Vol. 3, No. 4, pp. 184–195 (online), DOI: 10.1145/367177.367199 (1960).

[20] McCord, C.: Phoenix: A productive web framework that does not compromise speed and maintainability. (2014). https://phoenixframework.org.

[21] 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).

[22] NTT ドコモ: NTT ドコモ、「5G プレサービス」を 9 月 20 日（金曜）より開始 (2019). https://www.nttdocomo.co.jp/info/news_release/2019/09/18_00.html.

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

[24] Segan, S.: T-Mobile's LAA Creates Screaming Fast Speeds in NYC, *PC Magagine* (2018). https://www.pcmag.com/news/359649/t-mobiles-laa-creates-screaming-fast-speeds-in-nyc.

[25] Sholik, A.: Benchfella: Microbenchmarking tool for Elixir (2015). https://github.com/alco/benchfella.

[26] smartmobtech: Testing The First Ever 5G Network & Phone in USA (2019). https://smartmobtech.com/news/testing-the-first-ever-5g-network-phone-in-usa/.

[27] Turner, V., Gantz, J., Reinsel, D. and Minton, S.: The digital universe of opportunities: Rich data and the increasing value of the internet of things (2014). White Paper.

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

[29] 山崎　進，森　正和，上野嘉大，高瀬英希: 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，第 120 回情報処理学会プログラミング研究会, 熊本，情報処理学会プログラミング研究会 (PRO)，Vol. 2018, No. 2, 東京，p. (8) (2018). The paper and the presentation are available at https://zeam-vm.github.io/papers/GPU-SWoPP-2018.pdf and https://zeam-vm.github.io/GPU-SWoPP-2018-pr/, respectively.

[30] 山崎　進: ZEAM 開発ログ 目次 (2018). available at https://qiita.com/zacky1972/items/70593ab2b70d192813df.

[31] Yamazaki, S.: Hastega: Challenge for GPGPU on Elixir, *Lonestar ElixirConf 2019, Austin, TX, USA* (2019). The movie and the slides of this presentation are available at https://youtu.be/lypqlGlK1So and https://speakerdeck.com/zacky1972/hastega-challenge-for-gpgpu-on-elixir-at-lonestar-elixirconf-2019, respectively.

[32] Yamazaki, S.: Hastega: Challenge for GPGPU on Elixir (intend to apply it to machine learning) (2019). available at https://link.medium.com/ThF3Y8pbXU.

[33] 山崎 進，久江雄喜: SumMag : Elixir マクロのメタプログラミングを用いた並列プログラミング拡張機構 Hastega の解析部の設計と実装，情報処理学会論文誌プログラミング（PRO），Vol. 12, No. 3, pp. 7–7 (2019). https://ci.nii.ac.jp/naid/170000180471/.

[34] Yamazaki, S. and Hisae, Y.: Return of Wabi-Sabi: Hastega Will Bring More and More Computational Power to Elixir, *ElixirConf US 2019, Denver, CO, USA* (2019). The movie and of the slides of this presentation are available at https://youtu.be/uCkPyfFhPxI and https://speakerdeck.com/zacky1972/return-of-wabi-sabi-hastega-will-bring-more-and-more-computational-power-to-elixir, respectively.

表 1: 評価環境の共通特性

Table 1 Common features of evaluation environments

Elixir version	1.9.4
Erlang OTP version	22.1.5
Clang version	10.0.0
Clang revision	6c3fee47a6492b472be2d48cee0a85773f160df0

表 2: 各評価環境の特性

Table 2 Features of each evaluation environment

	iMac Pro	Ryzen
CPU maker	Intel	AMD
CPU model	Xeon W	Ryzen Threadripper 2990WX
Clock	2.3 GHz	3.0–4.2GHz
# of CPUs	1	1
# of physical cores on a CPU	18	32
SDRAM type	DDR4	DDR4
SDRAM clock	2666 MHz	2666MHz
SDRAM size	32GB	32GB
OS	macOS Mojave 10.14.6	Ubuntu 18.04
Compiler for Erlang	Apple clang 11.0.0	gcc 7.4.0

表 3: Pelemay バージョン 0.0.4 のロジスティック写像ベンチマーク実行結果

Table 3 The results of the Logistic Map benchmark using Pelemay version 0.0.4

	iMac Pro	Ryzen
CPU maker	Intel	AMD
CPU model	Xeon W	Ryzen Threadripper 2990WX
Clock	2.3 GHz	3.0–4.2GHz
Enum	1393.99 μ s/op	1064.00 μ s/op
Pelemay 0.0.4	582.67 μ s/op	536.34 μ s/op

表 4: 数の型と値の範囲に対する NIF 関数の返す結果

Table 4 The returned result value for the type and the range of a number

	float		LLONG_MIN	LONG_MIN	INT_MIN	0	INT_MAX + 1	UINT_MAX + 1	LONG_MAX + 1	ULONG_MAX + 1	LLONG_MAX + 1	ULLONG_MAX + 1
		~LLONG_MIN - 1	~LONG_MIN - 1	~INT_MIN - 1	～ -1	~ INT_MAX	~ UINT_MAX	~ LONG_MAX	~ ULONG_MAX	~ LLONG_MAX	~ ULLONG_MAX	～
enif_is_number	true	true	true	true	true	true	true	true	true	true	true	true
enif_get_int	false	false	false	false	true	true	false	false	false	false	false	false
enif_get_uint	false	false	false	false	false	true	true	false	false	false	false	false
enif_get_long	false	false	false	true	true	true	true	true	false	false	false	false
enif_get_ulong	false	false	false	false	false	true	true	true	false	false	false	false
enif_get_int64	false	false	true	true	true	true	true	true	true	true	false	false
enif_get_uint64	false	false	false	false	false	true	true	true	true	tru	true	false
enif_get_double	true	false	false	false	false	false	false	false	false	false	false	false