耐障害性が高くマルチコア性能を最大限発揮できる Elixir(エリクサー)を学んでみよう

北九州市立大学 山崎 進  京都大学 高瀬 英希 

fukuoka.ex

- 長らく組込みソフトウェア開発にはC言語が使われてきました
  - 最近ではC++やmruby，MicroPython も使われてきだしています
- とくに**C言語やC++には不満を覚えている**人も多くいらっしゃるのではないでしょうか
  - また，**例外処理やマルチコア対応に苦労している**方も少なくないでしょう

Elixir(エリクサー) は，最近登場した新しいプログラミング言語です

私たちは**組込みシステム/IoT用途にElixirが使える**と期待しています

C言語 のここがツライ。。。その1

並列処理は一苦労

- **C言語でマルチコア対応は大変!!**
- デッドロック・優先度逆転を防ぐよう注意して同期・排他制御をコーディングする必要あり
- 苦労してコーディングしても，そんなに性能が上がらない

```
void main_task(intptr_t exinf) {              /* メインタスク */
   FLGPTN flag;                          /* イベントフラグの待ち解除時の値を格納 */
   act_tsk(FLOW_TASK1); act_tsk(FLOW_TASK2);   /* タスクの起動 */
   wai_flg(FLOW_FLAG, 0x3, TWF_ANDW, &flag);   /* 各タスクが計算終わるのをイベントフラグで待つ */
}
void flow_task(intptr_t exinf) {              /* マルチコア計算タスク */
   int_t i, data_start, data_end;
   int_t tskno = (int_t) exinf;
   if (tskno == 1) {                      /* 配列のどの範囲を計算するかを決める */
      data_start = 0
      data_end = DATA_DIV
   } else if (tskno == 2) {
      ...
   }
   for (i=data_start; i<data_end; i++)        /* 実際の計算 (各要素を2倍する) */
      flow_data[i] *= 2;                  /* 本当はもしかするとここで排他制御が要るかもしれない */
   set_flg(FLOW_FLAG, tskno);                /* イベントフラグの対応するビットを立てる */
}
```

Elixir のここがイケている その1

並列処理が得意

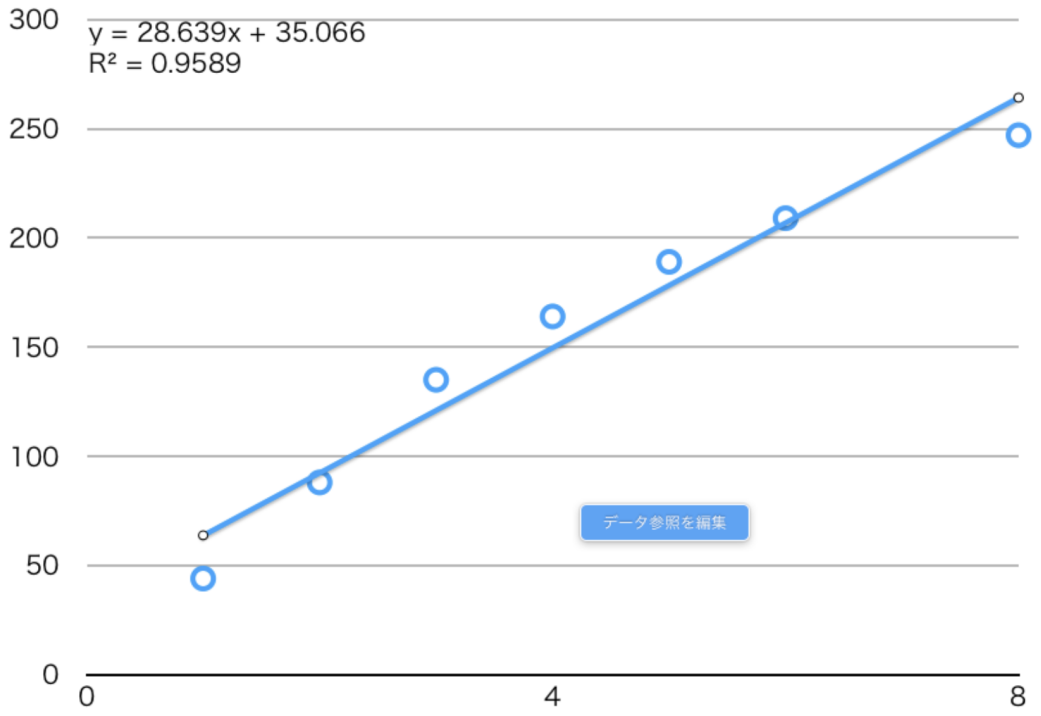
Elixirでマルチコアでの計算はシンプル

- **同期や排他制御を基本的に書く必要がない**
  - イミュータブル特性(変数の値が決まると不変になる特性)
  - 値の書き換えのために排他制御しなくていい
  - 各コアにコピーを置いておける
- **デッドロックも基本的に気にしなくていい**
  - 今はリアルタイム対応していないので優先度は設定できない
- **コア数が増えるにしたがって，きれいに性能が向上する**

```
0..10000                    # 0から10000までのリストを用意する
|> Flow.from_enumerable     # リストを各コアに分配する
|> Flow.map(& &1 * 2)       # 各要素を2倍する
|> Enum.to_list             # 計算結果をリストとして集める
```

コア数が増えるにしたがって，きれいに性能が向上する

1000/実行時間 をプロットしてみた



Elixir のここがイケている その2

シンプルでとっつきやすい,
それでいてパワフルな言語仕様

- **パイプライン演算子** |>
  - UNIXのパイプのように，計算結果を関数に次々と渡していける
- **MapReduceモデル**
  - Map: リストの各要素に関数を適用(並列で実行される)
  - Reduce: 計算結果を集約

```
0..10000
|> Flow.from_enumerable
|> Flow.map(& &1 * 2)                    # Map
|> Flow.map(& &1 + 1)
|> Enum.reduce(fn (x, acc) -> x + acc end)  # Reduce

# 全体として,リストの各要素を2倍して1加えてから総和を求める
```

Elixir のここがイケている その3

Ruby on Railsと同等以上の生産性で,
驚異の高レスポンス性能を誇るPhoenix

- Elixir の作者の José Valim は Ruby on Rails のコアコミッター
  - Elixir の文法やエコシステムは Ruby の影響を受けている
  - Phoenix も同様に Ruby on Rails の影響を受けている
  - つまり Phoenix は Ruby on Rails と少なくとも同等の生産性を誇る
  - Ruby や Rails にない洗練された機能を有するので同等以上かもしれない
- Elixir は Erlang VM で動作する
  - Elixir は Erlang の高い並列性と耐障害性を受け継ぐ
  - Phoenix は Elixir の性能を存分に発揮するように設計されている
- Phoenix の WebSocket は他の追従を許さない高速性・高レスポンス性を誇る

C言語 のここがツライ。。。その2

メモリ管理，例外処理は一苦労

- mallocは遅いしメモリリークするし，自分で管理するのがツライ
  - GCを扱えるライブラリは存在するが，不完全だしGC時に止まるので，組込みには向かない
- 例外処理のための記法が無く，悪名高きgotoやsetjmp/longjmpが登場してしまう

仮にJavaやmrubyなどであっても...

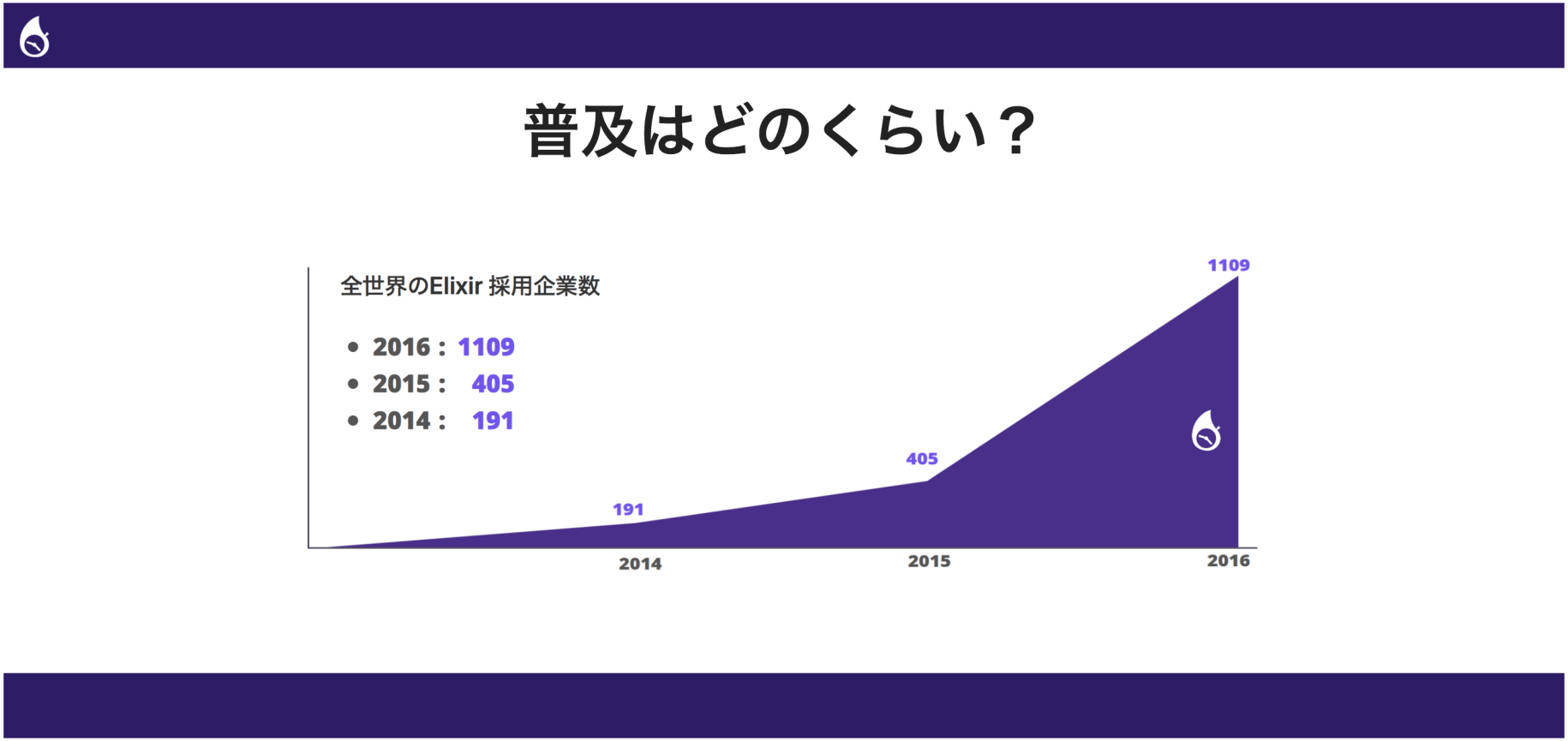
- GC時に止まってしまう
- メモリ障害が起こると，VM全体が落ちてしまう
- try/catchをきちんと書くのは，とても面倒

Elixir のここがイケている その4

プロセスごとの堅牢なメモリ管理で,
障害監視の仕組みが整っている

障害時に他に影響せず再起動OK!

例外処理に気を配らなくていい



普及はどのくらい？

全世界のElixir 採用企業数

- 2016 : 1109
- 2015 : 405
- 2014 : 191

IoTボードでの実績

Elixir は Linux が動作する IoT ボードでは動作させて，Groveモジュールの動作に成功した実績があります

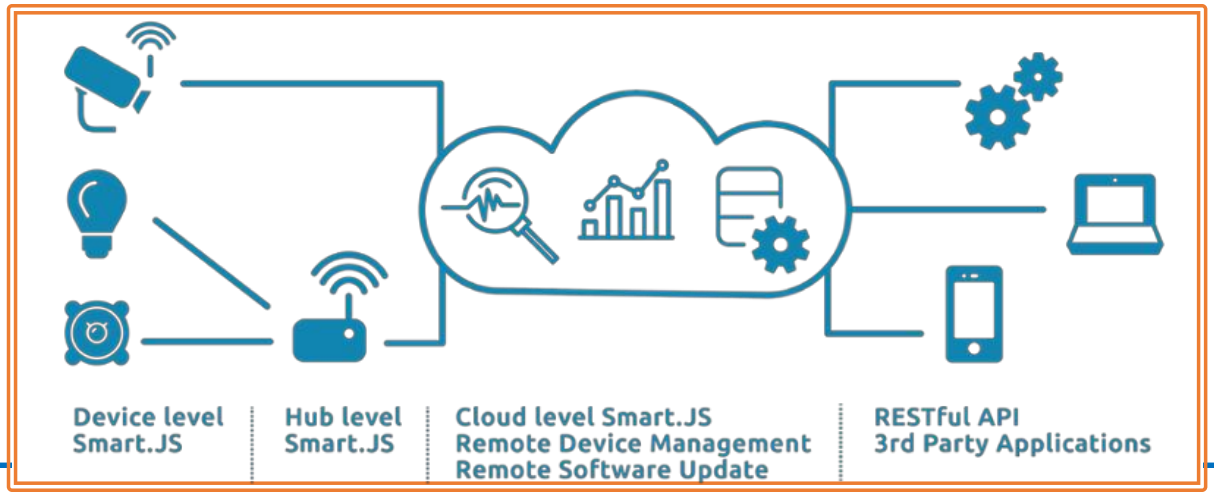
詳しくは高瀬先生  から...

ElixirでIoT!!

高瀬 英希
takase@i.kyoto-u.ac.jp

IoT: Internet of Things

- あらゆるモノやコトにヒトをインターネットに！
- デバイス／エッジ／クラウドの連携・集結で社会的価値を創出
  → 特にデバイスとエッジにて開発生産性の向上が不可欠



https://dzone.com/articles/what-is-an-iot-platform

組込み屋さんから見たElixir

- 読み書きしやすい／生産性が高い
- 関数型／並行処理プログラミング
- 分散システム対応／スケールしやすい
- 耐障害性が高い
- 軽量（なのかな？？）

IoTデバイスの開発にも
使えるんじゃね？？

fukuoka.ex でのこれまでの発表

#8：2018年 春のElixir入学式

『ElixirをIoTボードで動かしてみた
～ラズパイ・Zynq・big.LITTLE編～』

#11：DB／データサイエンスにコネクトするElixir

『環境センシングとデータ表示を
サクっと？やってみた』

Qiitaにて記事を公開中！！
https://qiita.com/takasehideki/items/8f43f1853ce88cbbe82e

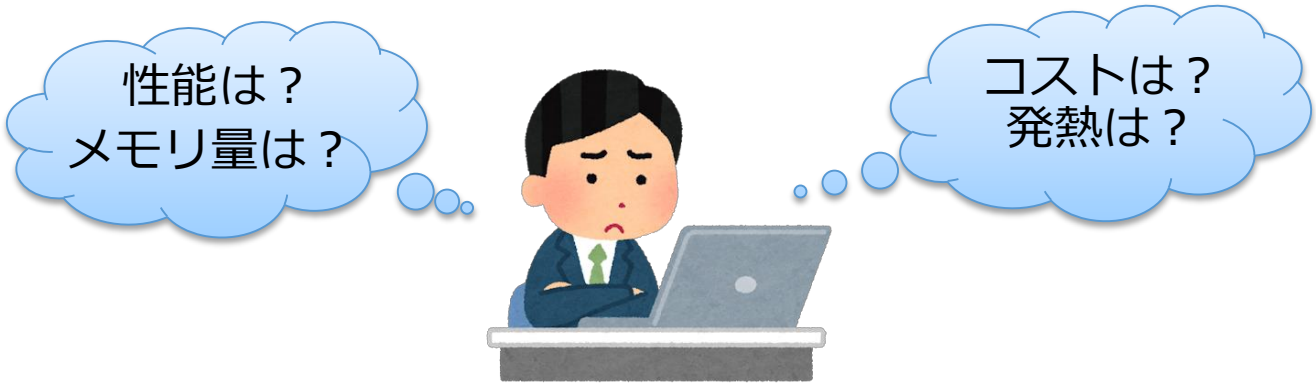
fukuoka.ex #8

ElixirをIoTボードで動かしてみた
～ラズパイ・big.LITTLE・Zynq編～

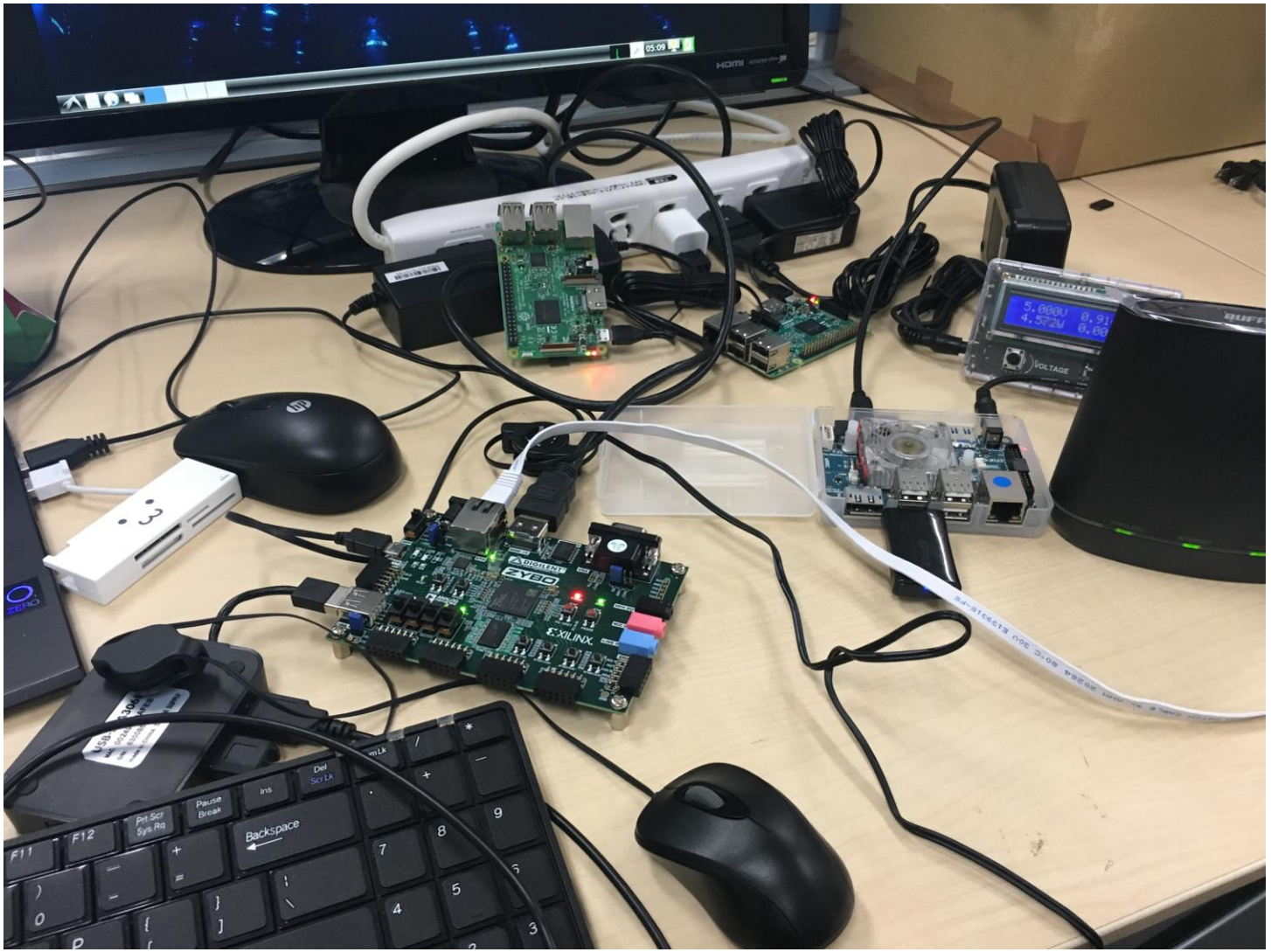
お題目

**ElixirをIoTボードで動かして
いろいろ調べてみよう**

今回のお話しのIoTボードの定義は
“きびきびとLinuxが動いて
さくっとネットに繋がる”モノ



選手のご紹介



Spec比較

- マイクロアーキとかOSの違いは？？

Board	OS	Core	Memory	Network
RPi3B-R	Raspbian 4.9	4x 1.2GHz Cortex-A53	1GB LPDDR2	150Mbps WiFi
RPi3B-U	Ubuntu 16.04			
ODROID-XU3	Ubuntu 16.04	4x 2.0GHz Cortex-A15 4x 1.4GHz Cortex-A7	2GB LPDDR3	300Mbps WiFi
ZYBO	Xillinux-2.0	2x 650MHz Cortex-A9	512MB DDR3	1GBit Ethernet
MacBook Pro	macOS 10.13.4	2x 3.3GHz Core i7 (4-threads)	16GB LPDDR3	433.3Mbps WiFi

評価用ツールキット

https://github.com/takasehideki/EEIoT

Evaluation toolkit for **El**ixir on I**oT** board

- 自動インストールスクリプト
- 評価環境設定スクリプト：コンパイル・ビルド等
- 基本的にはMIT Licenseですが,
  *.ex や git repos の著作権は各作者に帰属します

もうpullしてますよね！？？
starもください！！

評価用ツールキット

$ ./install.sh

- Elixir&Erlangをbrew, apt等で自動インストール
  - ✓OS環境を自動検知 (macOS, Raspbian, ubuntu, ...)
- cleanモード：パッケージインストールをremove
- sourceモード：ソースからErlang/Elixirをビルド
  - ✓ソースビルドはErlang OTP 20.3 / Elixir 1.6.5

$ ./setup.sh

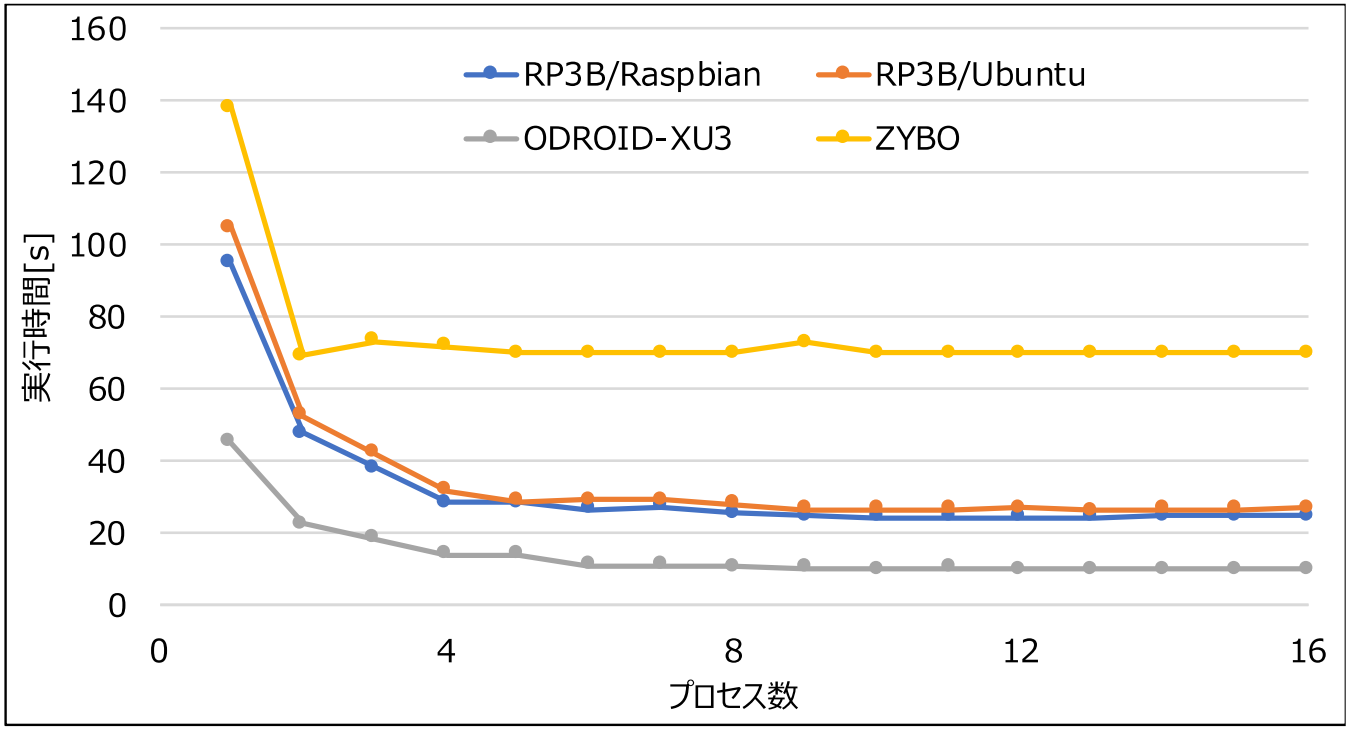
- *.exコンパイル，git repoのpullとビルド
- cleanモード：.beamとgit dirsを削除

用意したベンチマークアプリ

- ライプニッツ級数
  - 小数点演算の性能評価
- フィボナッチ数列（シングルプロセス版）
  - 整数演算およびスタック使用時の性能評価
- フィボナッチ数列の複数プロセス処理
  - 並列性能の評価
- 大規模データのストリーム処理
  - CSVデータの処理時の性能評価
- ロジスティック写像
  - 他言語モジュール(Rustler)を用いるNIF機能の性能評価
- Phoenixサーバの接続負荷
  - サーバのレイテンシとスループットの評価

結果抜粋：fib_multi

- fib_multi の並列プロセス数を変更
  - －RP3BとZYBOはコア数で並列性能が頭打ち
  - －ODROID-XU3では6並列で頭打ち（コア数は8）



結果抜粋：LogisticMap

[単位：秒]

Rustler不使用	1	2	4	8
RP3B/Raspbian	46.989	23.486	14.020	14.370
RP3B/Ubuntu	49.799	25.134	14.821	15.200
ODROID-XU3	23.884	12.726	8.895	6.865
ZYBO	87.184	2.957	41.789	48.387
Rustler使用	**1**	**2**	**4**	**8**
RP3B/Raspbian	24.318	21.584	13.745	14.550
RP3B/Ubuntu	24.065	21.550	14.613	14.948
ODROID-XU3	13.769	9.922	7.619	6.920
ZYBO	50.701	40.436	38.454	39.290

- Rustler不使用はbenchmarks3, 使用時はbenchmarks8
- 並列数が小さい場合はNIF機能が性能向上に寄与できる

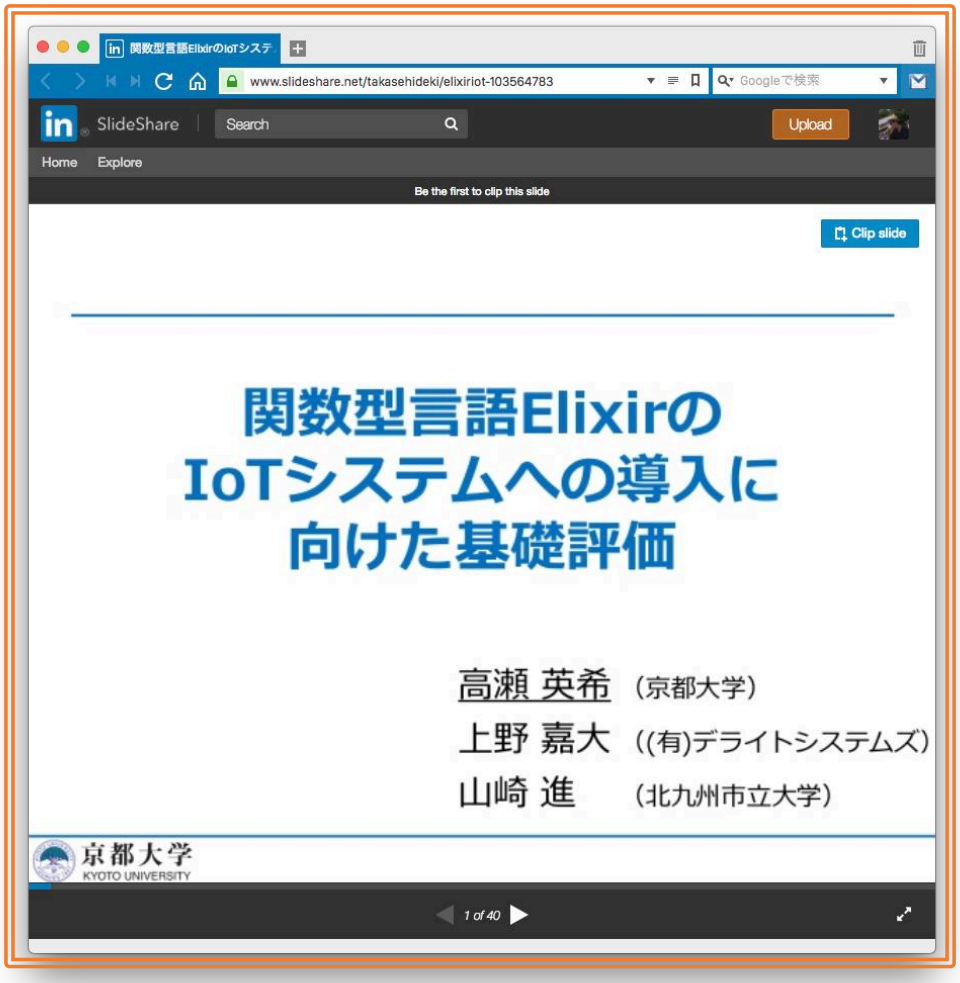
結果抜粋：phoenix-showdown

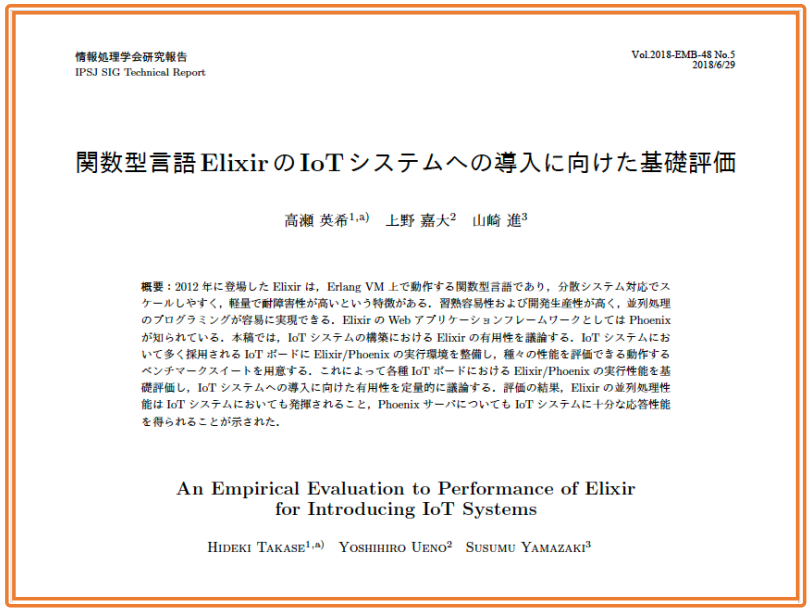
```
$ cd phoenix-showdown/phoenix/benchmarker/
$ mix phoenix.server
```

Board	Throughput [req/s]	Latency [ms]	Consistency [σms]
RP3B/Raspbian	785.25	351.47	771.85
RP3B/Ubuntu	878.89	113.30	31.68
ODROID-XU3	859.60	112.67	25.82
ZYBO	459.28	216.72	83.10

- ボード構成・接続形態よりもコア性能が重要とみられる
  - －ODROID-XU3はUSB-WiFiアダプタで接続
  - －ZYBOは有線接続だが性能を出すことはできない
- RP3BのOS間で明らかな差が見られる

残りはSlideShareと論文にて！





情報処理学会研究報告
IPSJ SIG Technical Report

Vol.2018-EMB-48 No.5
2018/6/29

関数型言語 Elixir の IoT システムへの導入に向けた基礎評価

高瀬 英希[1,a)]　上野 嘉大[2]　山崎 進[3]

概要：2012 年に登場した Elixir は，Erlang VM 上で動作する関数型言語であり，分散システム対応でスケールしやすく，軽量で耐障害性が高いという特徴がある．習熟容易性および開発生産性が高く，並列処理のプログラミングが容易に実現できる．Elixir の Web アプリケーションフレームワークとしては Phoenix が知られている．本稿では，IoT システムの構築における Elixir の有用性を議論する．IoT システムにおいて多く採用される IoT ボードに Elixir/Phoenix の実行環境を整備し，種々の性能を評価できる動作するベンチマークスイートを用意する．これによって各種 IoT ボードにおける Elixir/Phoenix の実行性能を基礎評価し，IoT システムへの導入に向けた有用性を定量的に議論する．評価の結果，Elixir の並列処理性能は IoT システムにおいても発揮されること，Phoenix サーバについても IoT システムに十分な応答性能を得られることが示された．

An Empirical Evaluation to Performance of Elixir for Introducing IoT Systems

Hideki Takase[1,a)]　Yoshihiro Ueno[2]　Susumu Yamazaki[3]

ベンチマークのまとめ

- Elixir並列処理はすごい！
  マイクロアーキよりコア数！！
  - XU3/A15は最大2.0GHzなのも効いている？
  - ZYBOは健闘はしてるかな，，，
  - なお消費電力は？？
    ZYBO << XU3 < RPi3 <<<(超えられない壁)<<< Mac
- サーバ性能はNIC形態が効く？
  - Ethernet接続で対決させたら違う傾向になりそう
- Elixirは最新版にしましょう
- バージョン毎・他言語／フレームワークの
  比較もしてみたら面白いのかなぁ

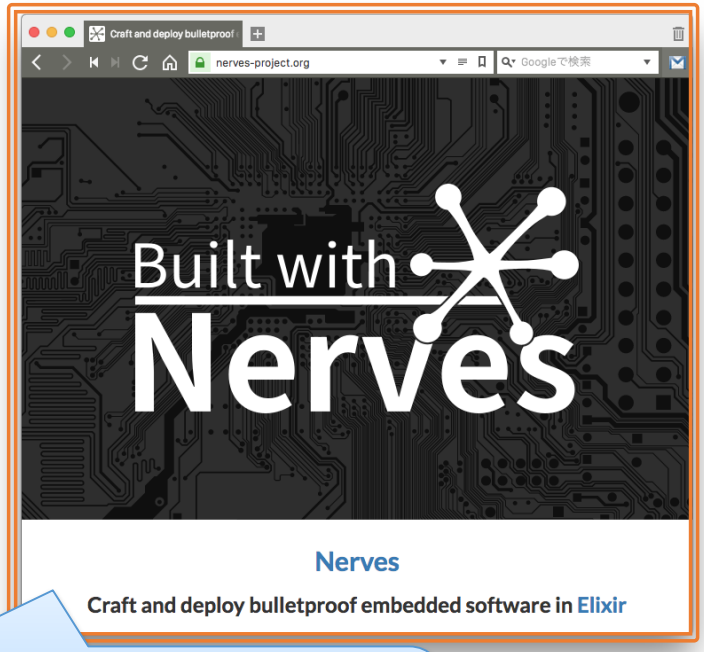
(#8のオチ)
ところでこれってIoT？？

fukuoka.ex #11

ElixirでIoT 第2回
「環境センシングとデータ表示をサクっと？やってみた」

高瀬 英希
takase@i.kyoto-u.ac.jp

世界の「ElixirでIoT」

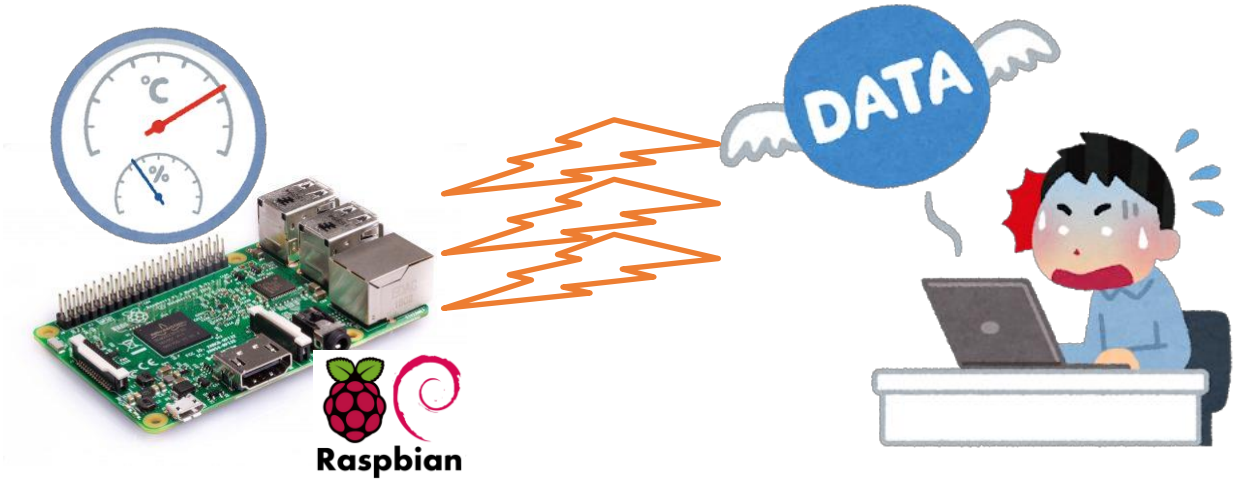


- ラズパイ等が(限定)対象
- メモリサイズ数10MB！
- クロス開発はツラい,,,

- HWもSWもセットでご提供!!
- Erlang VMがbare metalで動く
- 300MHz Cortex-M7 & 64MB Flash
- On-board WLAN & Pmodコネクタ

今回のお題目

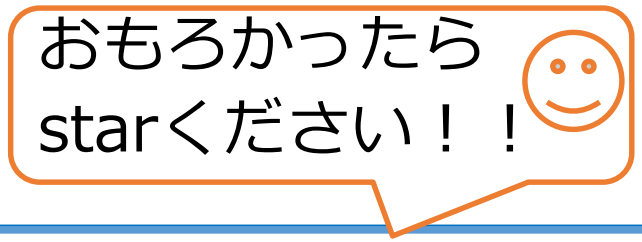
Elixirとラズパイ3Bで
環境データをセンシングして
表示させてみよう



IoTに限らないElixir/Phoenixアプリ開発の
TIPS(a.k.a.ハマりどころ)も共有します

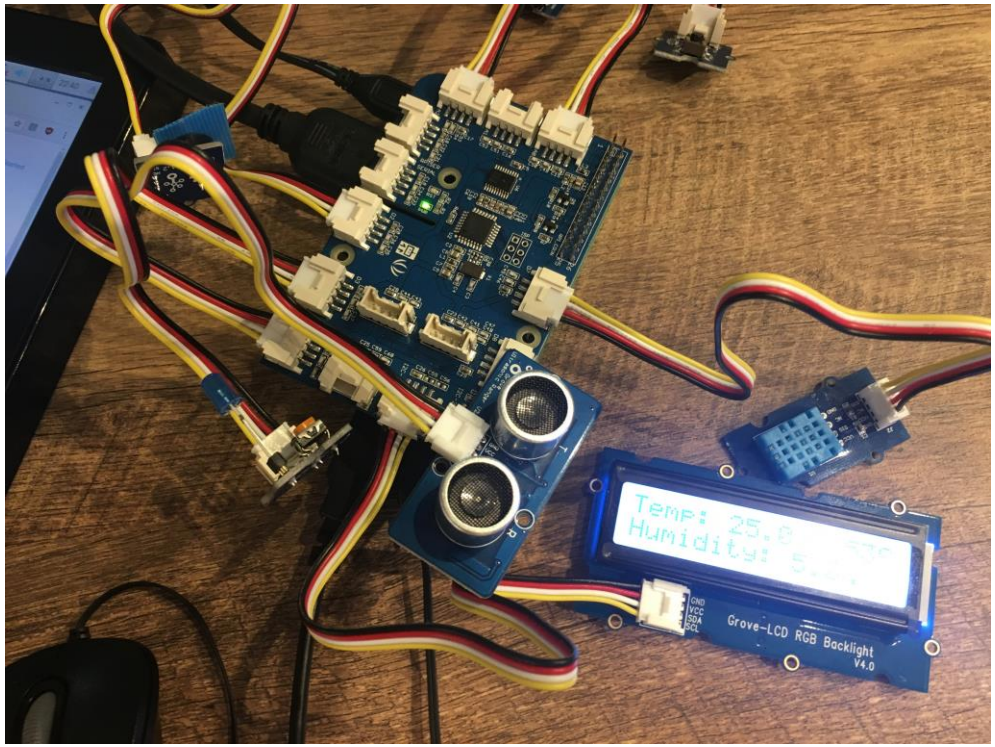
やったこと

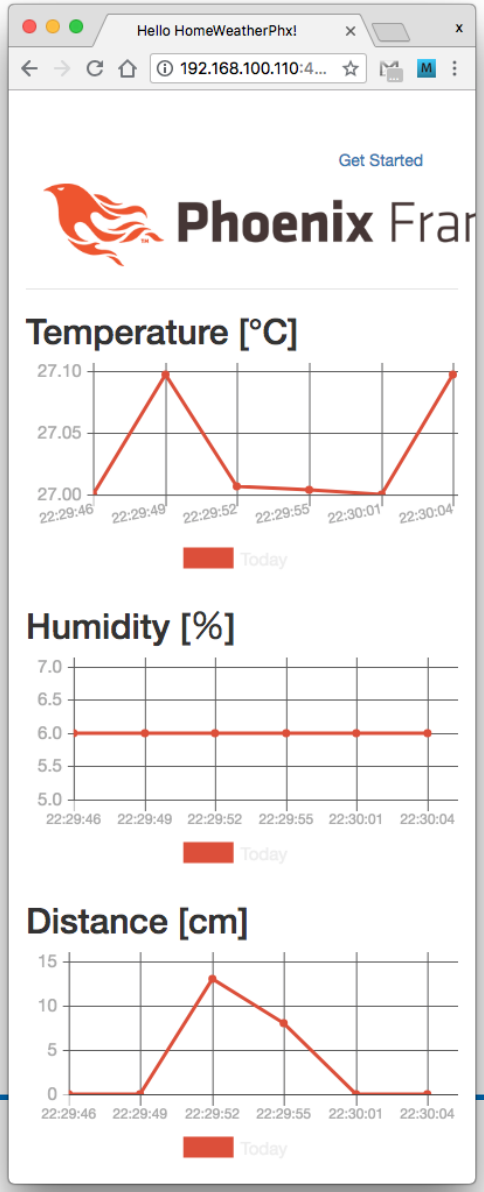
1. GrovePiライブラリを調べて試してみた
2. 温湿度センサDHT11を使ってみた
3. LCD displayにデータを表示してみた
4. センサデータをCSVで書き出してみた
5. Phoenixページにグラフを表示してみた
6. 書き出したCSVデータをグラフ表示してみた
7. **環境センシングしながらリアルタイムでグラフ表示してみた！**



GitHubのディレクトリ対応：
1. grovepi_example/
2. dht_sensor/
3. home_weather_dislay_host/
4. home_weather_csv/
5. phx_chartjs/
6. phx_csvchart/
7. home_weather_phx/

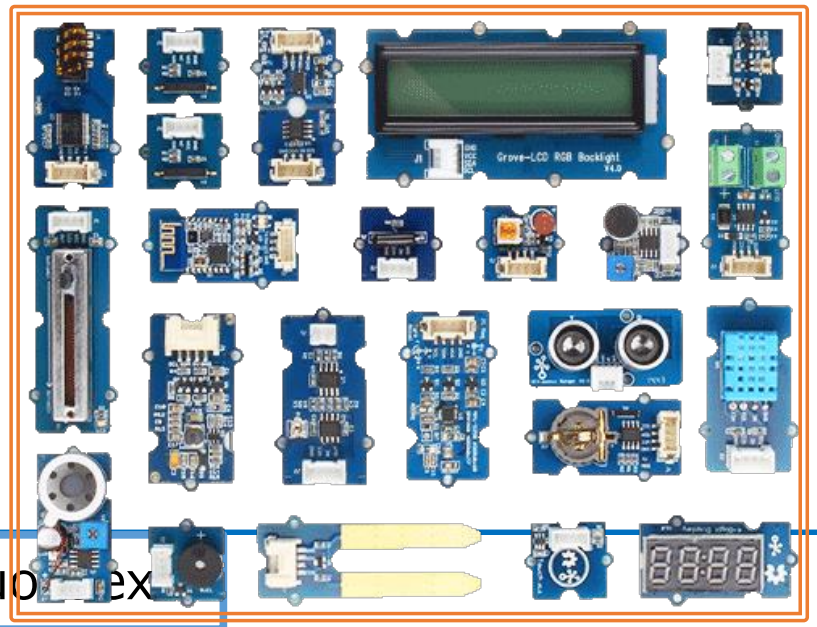
論より証拠！


Temp: 25.8
Humidity: 5
Grove-LCD RGB Backlight
V4.0


Hello HomeWeatherPhx!
192.168.100.110:4...
Get Started
Phoenix Fran
Temperature [°C]
27.10
27.05
27.00
22:29:46
22:29:49
22:29:52
22:29:55
22:30:01
22:30:04
Today
Humidity [%]
7.0
6.5
6.0
5.5
5.0
Today
Distance [cm]
15
10
5
0
Today

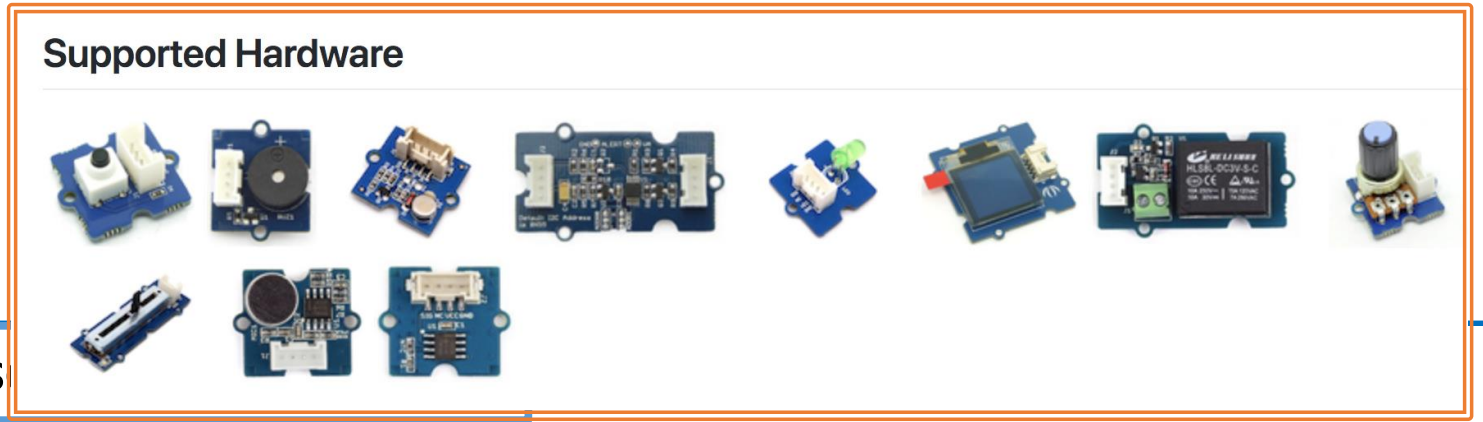
1. GrovePiライブラリ

- Groveとは？：
  - IoTのセンサ・アクチュエータが画一化されたモジュール
    - ✓最低でも86種類？（@mine820さんのQiita記事調べ）
  - 入出力ピンx4のGroveコネクタで簡単に付け替えできる
  - ラズパイやArduinoなどに適合するシールドもあり
    → IoTシステムをお手軽にラピッドプロトタイピング！



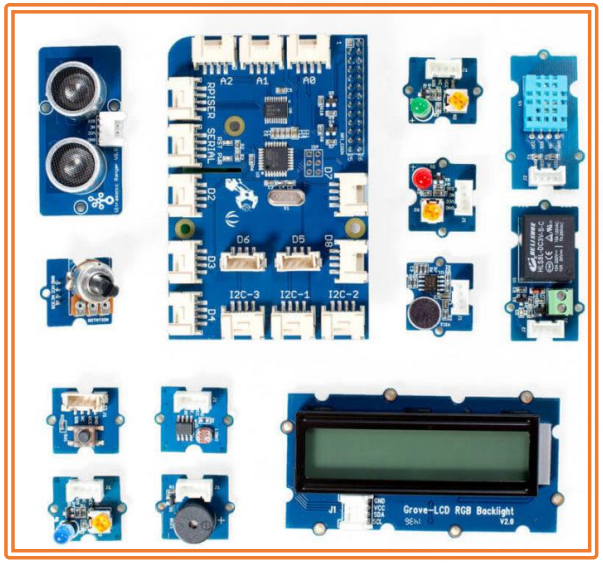
1. GrovePiライブラリ

- nerves_grove
  - Elixirで使えるGroveライブラリ from Nerves
  - ブレッドボードで稲作の必要あり
  - 下回りは elixir_ale (Elixir Actor Library for Embedded)
    - ✓ラズパイ用のGPIOs, I2C, SPIのライブラリ
    - ✓MIX_ENV=prod で動かす必要あり？
    - ✓さらにその下は GenServer/SuperVisor
  - 2016年9月で開発止まってる，，，



1. GrovePiライブラリ

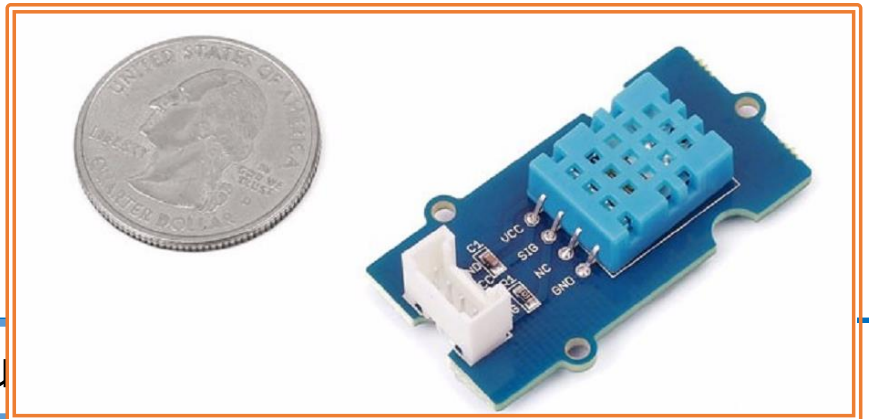
- GrovePi
  - ラズパイのGrovePi+シールドに対応したライブラリ
    - ✓田んぼを耕さなくていい！
  - examples も用意されていて親切！
    - ✓alarm/
    - ✓demo_pivotpi/
    - ✓demo_rgblcd
    - ✓home_weather_display/
      - Nervesアプリ,,,
    - ✓led_fade/



Qiita「ElixirでIoT#5」にて
導入方法などを解説してます!!

2. 温湿度センサ

- Temperature&Humidity Sensor
  - －DHT11を搭載したデジタル温度湿度センサ
    - ✓40ビットのビットフィールドで取得される
      - 温度：0.0~50.0℃ / 1.0℃単位 / ±2.0℃誤差
      - 湿度：20.0~90.0% / 1.0%単位 / ±5.0%誤差
  - －Elixirからの使い方はソース嫁 m(_ _)m
    - ✓3秒毎に値の差分があれば標準出力
    - ✓ホスト上(nerves_runtime無し)で動くようにしてます



3. LCDモジュールに表示

- LCD RGB Backlight
  - 16x2の英数字が出力可能
    - ✓API／サンプルが豊富で使いやすい！
  - バックライトがRGBで制御できる（カッコイイ

4. センサデータをCSVに書き出し

- DHTから取得した温度湿度データをCSVで保存
- Timexで現在時刻も取得 ref. @kobataro, @piacere

home_weather_csv/lib/home_weather_csv.ex

```
# Create CSV file
  File.write "dhtdata.csv", ""
  …
  # Get date
  date = Timex.now("Asia/Tokyo")
    |> Timex.format!( "%Y-%m-%d %H:%M:%S", :strftime )
  # Write data to CSV
  File.write "dhtdata.csv", "#{date},#{temp},#{humidity}¥n", [:append]
  #File.write "../phx_csvchart/priv/static/dhtdata.csv",
  #         "#{date},#{temp},#{humidity}¥n", [:append]
  …
```

Tips: 現在時刻の取得と変換

Tips: ファイル位置は任意でもOK
(プロジェクトトップからの相対パス)

5. Phoenixにグラフ表示

- Chart.js を使用 (日本語ページもあり)
  －オフラインでも使えるはず？？

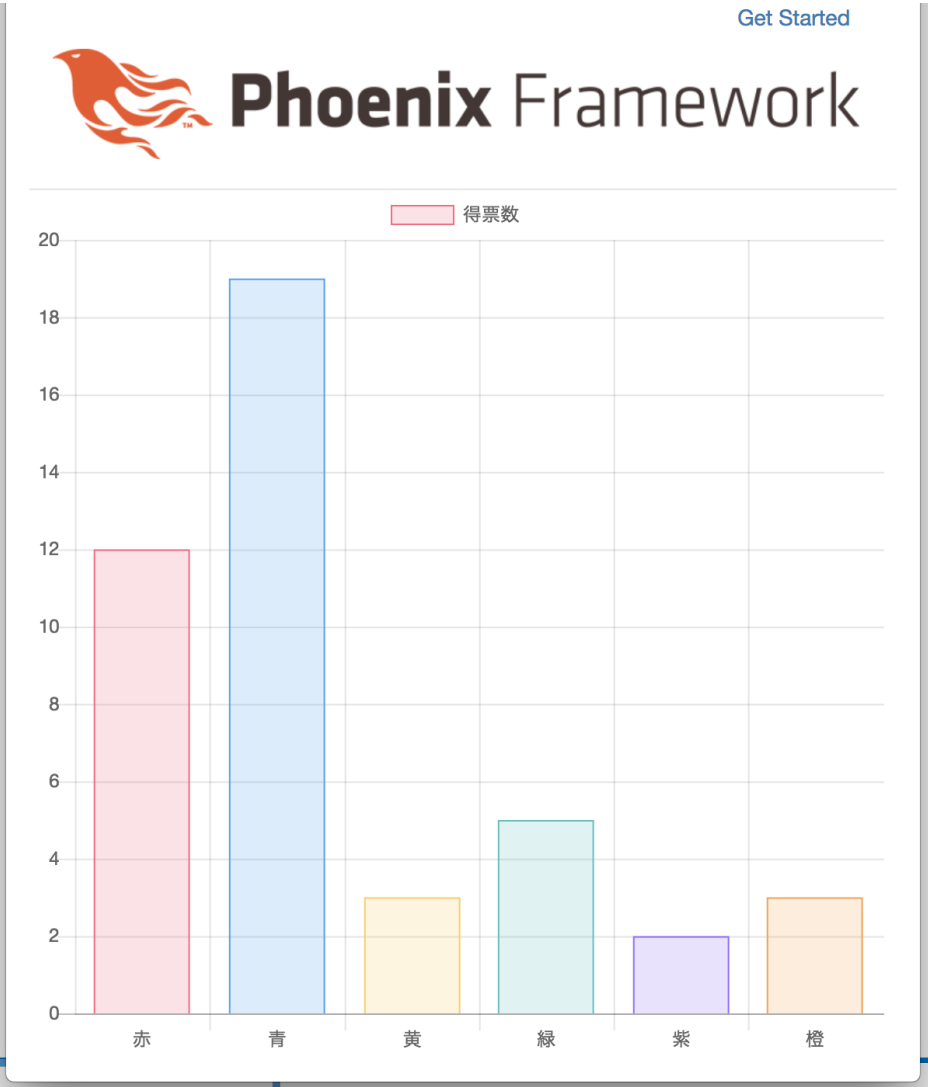
phx_chartjs/lib/phx_chartjs_web/templates/page/index.html.eex

```
<script src="https://cdnjs.cloudflare.com/ajax/libs/Chart.js/2.7.2/Chart.min.js">
</script>
<canvas id="myChart" width="400" height="400"></canvas>

<script>
var ctx = document.getElementById("myChart");
var myChart = new Chart(ctx, {
    type: 'bar',
    data: {
    …
```

Tips: Chart.jsのURLを直接指定じゃないとPhoenixエラー？

5. Phoenixにグラフ表示



6. CSVをグラフ表示

- 4. home_weather_csv/ で書き出したデータをChart.jsでグラフ表示
  - $ mix phx.new phx_csvchart --no-ecto --no-brunch
  - phx_csvchart/prliv/static/dhtdata.csv にあるとする
  - ソースは「アルゴリズム雑記」よりほぼパクリ^^;

phx_csvchart/lib/phx_csvchart_web/endpoint.ex

```
plug Plug.Static,
    #at: "/", from: :home_weather_phx, gzip: false,
    #only: ~w(css fonts images js favicon.ico robots.txt)
    at: "/", from: :home_weather_phx, gzip: false
```

Tips: Static Fileをサーブする

6. CSVをグラフ表示

- TIPS: MIX_ENV=prod でPhoenix立ち上げ
  −$ cp config/dev.exs config/prod.exs して編集する

config/prod.exs

```
config :phx_csvchart, PhxCsvchartWeb.Endpoint,
  http: [port: 4000],
  debug_errors: true,
  code_reloader: false,
  check_origin: false,
  server: true,
  watchers: [],
  secret_key_base: System.get_env("PhxCsvchart_SEC_KEY_BASE")
```

Tips: prod だとreloadできない

7. リアルタイムに！

- 1つのElixir/Phoenixプロジェクトに統合
  - home_weather_phx/lib/home_weather_phx/application.ex と home_weather_phx/lib/home_weather_phx.ex にGrovePiアプリ記述を追加
    - ✓無駄に超音波センサも追加

MIX_ENV=dev でも
動くんかい！

config/dev.exs

```
config :home_weather_phx, HomeWeatherPhxWeb.Endpoint,
  live_reload: [
    patterns: [
      ~r{priv/static/.*(js|css|png|jpeg|jpg|gif|svg|csv)$},
```

Tips: phoenix_live_reloadで
CSV変更時にページを自動更新

まとめ

- 環境センシングとデータ表示をサクっと？やってみた
  - ✓「サクッと？」だったかどうかは議論の余地あり^^;
  - －GrovePi: nerves_grove + elixir_ale, deps
  - －Phoenix: live_reload, prod.exs, priv/
- 「ElixirでIoT」の良さと課題
  - －Phoenix連携はエッジサーバ向けにも良い！
  - －GenServer/Supervisor で polling だけはツラい
  - －リアルタイム性の保証なんてあかんでしょうね，，，
  - －相も変わらずLinuxは必須です
- fukuoka.ex の ecosystem はいいよ！！
  - － Very Thanks!! @piacere, @enpedasi, @tsuchro, @kobataro

次回予告？？

fukuoka.ex #XX

ElixirでIoTロボットを開発してみた
～ROS通信・FPGA編～

協力者を募集中！
Elixirで新しい世界へ！！

IoTボードでの実績

Elixir は Linux が動作する IoT ボードでは動作させて，Groveモジュールの動作に成功した実績があります

詳しくは高瀬先生  から...

Elixirの組込みソフトウェア開発対応

- 残念ながら Elixir は現在では省メモリ・リアルタイム対応していません
- 今すぐElixirを組込みソフトウェア開発に応用できるというものではありません
  - IoTボードや IoT バックエンドサーバーに適用する場合を除く
- しかしながら，私たちは現在，ZEAM という言語処理系を開発しています
  - ZEAM の狙いの1つは，Elixir で組込みシステムを開発できるようにしようとしています

micro Elixir / ZEAM

超高速リアルタイム処理にチューニングしたElixir処理系

- 最適化しやすいように言語仕様をチューニング
- LLVM を拡張した Polymorphic LLVM によるインラインアセンブリコード記述
- 組込み/IoT用途: 省メモリ，互換性，ハードリアルタイム
- ネイティブなコールバック方式のマルチタスク機構 Sabotender
- アノテーションによるマルチコアCPU/GPU活用 Hastega
- 高耐障害性のプロセス独立メモリ管理機構
- 大域的コード最適化機構
- ネイティブコードに限りなく近い高速性能

micro Elixir / ZEAM
最適化しやすいように言語仕様をチューニング

- Elixir の言語仕様やライブラリの中で，オーバースペックで最適化しにくいものを最初はカットする
  - 整数型の値に上限がない → 符号つき64ビット整数値

micro Elixir / ZEAM
LLVM を拡張した Polymorphic LLVM による インラインアセンブリコード記述

- 最近のコンパイラのコード生成系には LLVM が採用されている
- 中間言語 LLVM IR は，C言語とアセンブリ言語の中間のようなコード体系

64ビット整数型のローカル変数 a に定数1を代入するコード

```
%a = alloca i64
store i64 1, i64* %a
```

- Elixir では型多相(polymorphic type)，つまり例えば足し算のコードは整数型でも浮動小数点数型でも同じコードで動作する
- そこで，LLVM を型多相に拡張した Polymorphic LLVM を提案し，これを用いてインラインアセンブリコードを記述できるようにする

```
def add(a, b) do
  asm %ret = alloca
  asm %1 = load *%a
  asm %2 = load *%b
  asm %3 = add %1, %2
  asm store %3, *ret
  asm %ret
end
```

micro Elixir / ZEAM
LLVM を拡張した Polymorphic LLVM による インラインアセンブリコード記述

- 型推論を行って，想定される型ごとに特化したコードを生成する

```
def add(a, b) do          # 4. 引数・戻り値ともに整数型と浮動小数点数型がありえる
  asm %ret = alloca
  asm %1 = load *%a       # 2. %a は整数型と浮動小数点数型がありえる
  asm %2 = load *%b       # 2. %b は整数型と浮動小数点数型がありえる
  asm %3 = add %1, %2     # 1. %1,%2,%3は整数型と浮動小数点数型がありえる
  asm store %3, *ret      # 3. *ret は整数型と浮動小数点数型がありえる
  asm %ret
end
```

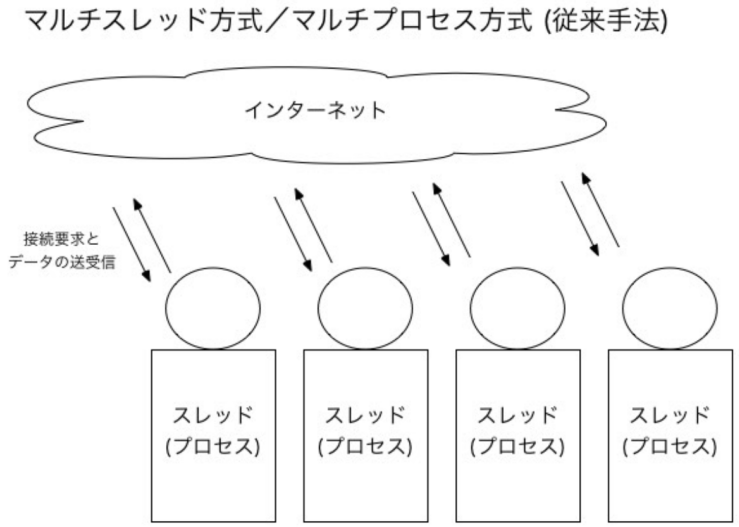
- アセンブリコードベリファイアにより，違法なメモリアクセスをしていないかを検査する
  - store のポインタアクセスは，alloca で確保したメモリ領域なので，問題ない
- 型安全性を破らないように，関数の先頭に引数の型に合わせて分岐するコードを挿入する
  - a,b がともに整数型の場合
  - a,b がともに浮動小数点数型の場合→addをfaddに
  - a が整数型，bが浮動小数点数型の場合→aを浮動小数点数型にキャスト
  - a が浮動小数点数型，bが整数型の場合→bを浮動小数点数型にキャスト
  - a, b のどちらかが，整数型・浮動小数点数型のどちらでもない型の場合→型エラー

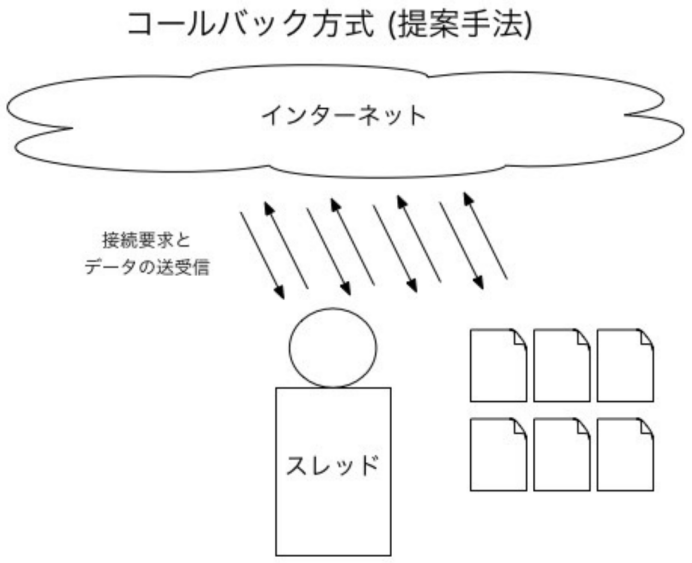
micro Elixir / ZEAM
組込み/IoT用途: 省メモリ，互換性，ハードリアルタイム

- 組込みシステム開発に欠かせないのが省メモリ・リアルタイム性
- 既存資産を活用するために互換性も確保したい
- 省メモリのためのフィーチャ
  - リスト構造を，要素の型によって分類
  - バイト配列・ビット配列
- 互換性のためのフィーチャ
  - C言語のヘッダファイルを読み込んで Polymorphic LLVM で扱えるようにする
  - エンディアンに関しても型多相になるようにする
- リアルタイム性のためのフィーチャ
  - 処理ごとに優先度を指定できるようにする
  - アセンブリコードからコード実行にかかる所用実行時間を推定
  - 処理ごとに指定されたデッドラインをもとにスケジューリングを最適化

micro Elixir / ZEAM ネイティブなコールバック方式のマルチタスク機構 Sabotender

- 我々の予測ではIoTの急拡大等で2020年の後にはインターネットのデータ流通量が現在の200〜500倍になる
- 従来のマルチプロセス/スレッド方式では，接続するたびに数MBのスタックメモリが必要なので，アクセスが集中するとメモリを使い果たして固まってしまう
- Node.jsで採用されたコールバック方式は，スタックメモリを最小限に抑えられるので，同時セッション数やレスポンスを大幅に改善できる





micro Elixir / ZEAM ネイティブなコールバック方式のマルチタスク機構 Sabotender

コールバック方式の原理

```
void blink_led() {
    turn_on(led1);
    sleep(500);
    turn_off(led1);
    sleep(500);
}
```

- 下記のようにコードを変換する
- sleep の後で第2引数で指定された関数をコールバックする

```
void blink_led_1() {
    turn_on(led1);
    sleep(500, blink_led_2);
}

void blink_led_2() {
    turn_off(led2);
    sleep(500, blink_led_3);
}

void blink_led_3() {}
```

micro Elixir / ZEAM ネイティブなコールバック方式のマルチタスク機構 Sabotender

今までに次のような成果が得られた

- Zackernel: NodeプログラミングモデルのC++実装
- LCB: NodeプログラミングモデルのElixir実装
- 1スレッド/プロセスあたりのメモリ消費量の比較:
  - Zackernel vs C++11スレッド: 1スレッドあたり**204バイト** / 約546KB
  - LCB vs Elixirプロセス: 1スレッド/プロセスあたり**1332バイト** / 2835バイト
- 処理系レベルから設計を見直すことで，LCBをZackernel並みのメモリ消費量に抑える最適化を施せる余地があるのかもしれない
  - これを **Sabotender** と称する

micro Elixir / ZEAM アノテーションによる マルチコアCPU/GPU活用 Hastega

- AIの社会実装が今後ますます増加する
- 我々の予測ではIoTの急拡大等で2020年の後にはインターネットのデータ流通量が現在の200〜500倍になる
- AIの処理能力を高める必要がある
- そこで，Elixirのマルチコア CPU/GPU活用について研究を進めた

micro Elixir / ZEAM アノテーションによる マルチコアCPU/GPU活用 Hastega

GPUの特徴

- コア数が桁違いに多い
  - Intel Xeon: 論理コア数〜20数個
  - NVIDIA GeForce GTX 1080 Ti: **3000個以上！**
- **SIMD** (単一命令列，複数データ列)アーキテクチャ
- 次のような場合に高速に計算できる
  - **単純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

micro Elixir / ZEAM アノテーションによる マルチコアCPU/GPU活用 Hastega

- 次のような場合に高速に計算できる
  - **単純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

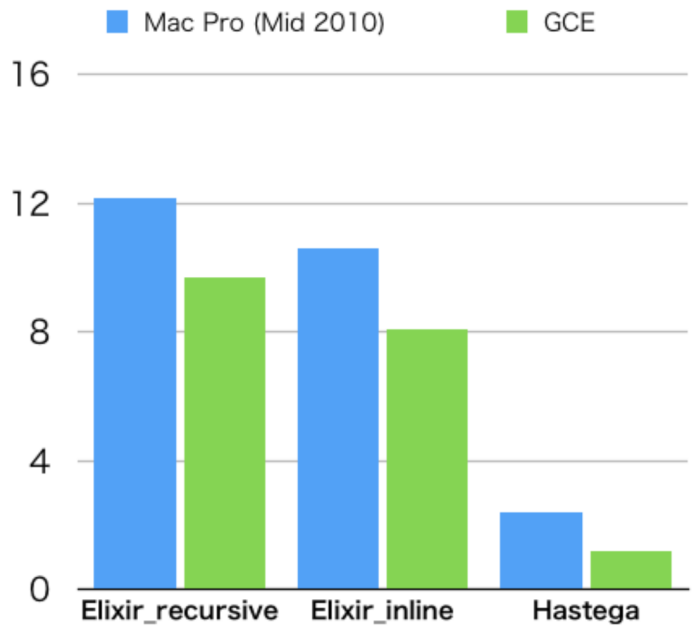
```
0..10000          単純で均質で大量にあるデータ
|> Enum.map(foo)  同じような命令列
|> Enum.map(bar)
```

単純な方法でGPUを駆動するコードにできる！

```
__kernel void calc(
  __global long* input,
  __global long* output) {
  size_t i = get_global_id(0);
  long temp = input[i];
  temp = foo(temp);       # 実際には関数呼出しをインライン展開することで,
  temp = bar(temp);       # さらなる最適化を促進する
  output[i] = temp;
}
```

Elixirからの速度向上

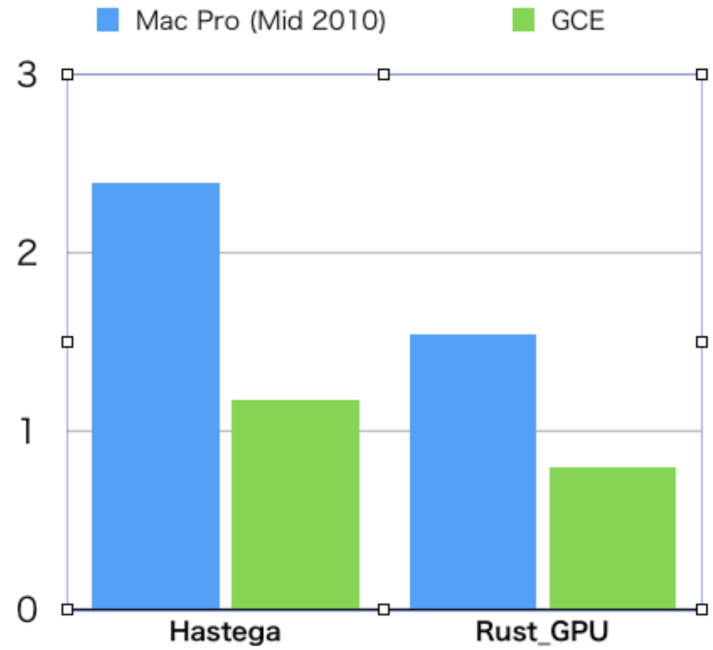
提案手法はElixir単体のコードと比べて4.43〜8.23倍高速になった



	Mac Pro	GCE
Elixir_recursive	12.177	9.674
Elixir_inline	10.579	8.075
Rustler_GPU (Hastega)	2.388	1.176

ネイティブコードとの比較

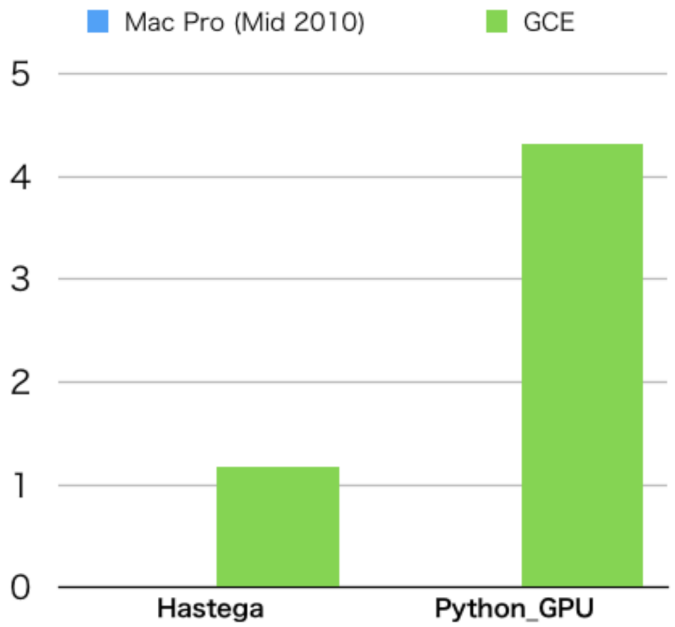
- 提案手法はGPUを使用するネイティブコードと比べ，1.48〜1.54倍遅くなっただけである
- これはErlang VMのオーバーヘッドで，**潜在的な最適化の余地**であると考えられる



	Mac Pro	GCE
Rustler_GPU (Hastega)	2.388	1.176
Rust_GPU	1.546	0.797

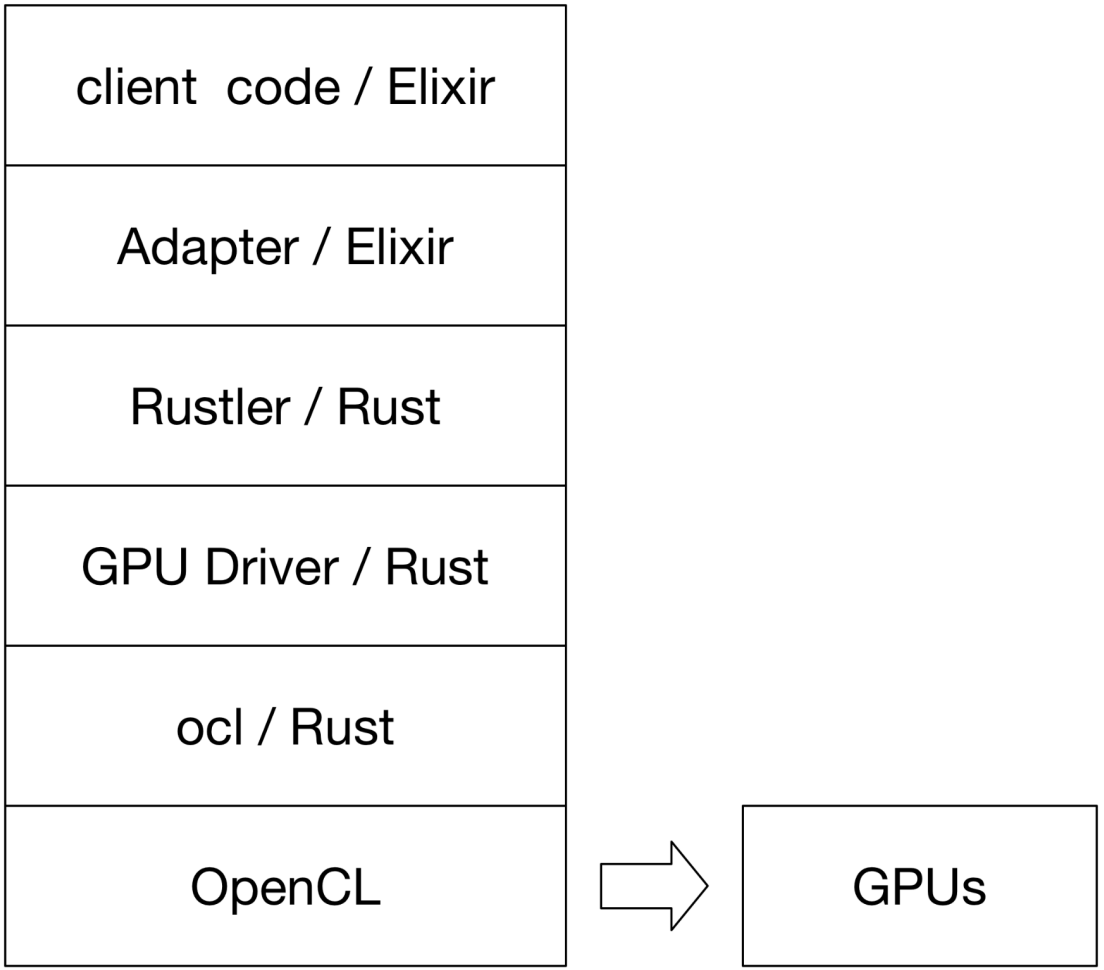
Pythonからの速度向上

提案手法はGPUを使用するPythonのコードと比べ，3.67倍高速である



	GCE
Rustler_GPU (Hastega)	1.176
Python_GPU	4.316

micro Elixir / ZEAM アノテーションによる マルチコアCPU/GPU活用 Hastega



micro Elixir / ZEAM アノテーションによる マルチコアCPU/GPU活用 Hastega

- Hastega(ヘイスガ): Elixirのコードをマルチコア CPU/GPU向けに最適化するサブシステム
- 当初はCPUで実行するかGPUで実行するかはアノテーションで指定する
- 将来的には自動で負荷分散するようにしたい

micro Elixir / ZEAM
高耐障害性のプロセス独立メモリ管理機構

- Elixir / Erlang VM の優れた特性である耐障害性を micro Elixir / ZEAM でも受け継ぐ
- プロセスごとに独立したメモリ管理(GCを含む)を採用する
- できる限りスタック領域にオブジェクトを配置する
- 一定サイズ以上の巨大なオブジェクト，プロセスで共有するオブジェクトは，ページ単位で共有メモリ領域に配置し，領域ごとまとめてGC管理する
  - イミュータブル(値が決まったら不変)だからこそ出来る簡便かつ高速なメモリ管理
  - ページ単位なのでフラグメントが起こらない
- 戻り値は次のように扱う
  - レジスタにおさまるならばレジスタで渡す
  - 一定サイズ未満であればスタック領域で渡す
  - 一定サイズ以上であれば共有メモリ領域で渡す

micro Elixir / ZEAM 大域的コード最適化機構

- GPUの最適化をしようと思ったら，ライブラリの関数の意味を汲み取って大域的に最適化する必要がある

```
0..10000
|> Enum.map(foo)
|> Enum.map(bar)
```

単純で均質で大量にあるデータ

同じような命令列

ライブラリEnumやFlowのレベルで最適化

- そこで，ライブラリごとにコンパイラの構文木拡張と意味解析器，最適化器を記述できる仕組みを導入する

micro Elixir / ZEAM
ネイティブコードに限りなく近い高速性能

今まで紹介したようなフィーチャーを活用して，抜本的な最適化を研究し，
アセンブリ言語でガチガチにチューニングしたネイティブコードに遜色ない高速性能を目指す

fukuoka.ex

「fukuoka.ex」は「膨大なデータやアクセスの高速処理」と「高い開発効率」を両立できる並列・分散プログラミング言語「Elixir（エリクサー）」と，そのWebフレームワーク「Phoenix（フェニックス）」を福岡で広めるコミュニティです。

Twitter で Elixir / Phoenix についてつぶやくと，漏れなくレスポンスが来ます！

https://fukuokaex.fun

fukuoka.ex

「fukuoka.ex」は「膨大なデータやアクセスの高速処理」と「高い開発効率」を両立できる並列・分散プログラミング言語「Elixir（エリクサー）」と，そのWebフレームワーク「Phoenix（フェニックス）」を福岡で広めるコミュニティです。

Twitter で Elixir / Phoenix についてつぶやくと，漏れなくレスポンスが来ます！

https://fukuokaex.fun

「*.ex」：みなさんの街にも Elixir コミュニティを作りましょう！