# スーパーコンピュータが組込みシステムに降りてくる！
## ～新時代の高性能組込みシステムの SIMD／ベクトル処理の要点を押さえる

北九州市立大学 山崎 進

© 2021 Susumu Yamazaki

# 背景

- スーパーコンピュータで培われてきた並列計算の技術が組込みシステムに降りてきたのは,
  今に始まったことではない
  - 1994年発売の初代PlayStation (右図)
  - 2000年発売のPlayStation 2


©MarcelBuehner, Creative Commons

© 2021 Susumu Yamazaki

# 背景

- 2021年3月発表 次期ARMアーキテクチャとなるArmv9アーキテクチャ
  - スーパーコンピュータ富岳で用いられるベクトル命令SVE/SVE2が採用
- RISC-V
  - ベクタ拡張を搭載したD1 chip搭載のIoTボードが市販
  - 2021年6月 ベクタ拡張を搭載したRISC-Vチップを含む高性能RISC-VチップをIntelが作ると発表

© 2021 Susumu Yamazaki

# 背景

- SIMD(Single Instruction Multiple Data) についてはこれに先んじて普及
  - 広く普及している現行のArmv8アーキテクチャにSIMDが搭載
- GPUも基本的にはSIMDアーキテクチャ
  - Arm MaliやNVIDIA JetsonのようにGPUを搭載したIoTも一般的

© 2021 Susumu Yamazaki

# ハイエンドの組込みシステムに搭載される SIMD／ベクトル処理機能を活用するには？

© 2021 Susumu Yamazaki

# ハイエンドの組込みシステムに搭載されるSIMD/ベクトル処理機能を活用するには

- 一般的な答え
  - スーパーコンピュータで長年培われてきた技術に由来するBLAS/LAPACKという線形代数に基づくライブラリと，それを活用するOSSライブラリを利用しましょう
- その理由
  - BLAS/LAPACKは，人知のかぎりを尽くしてSIMD／ベクトル処理機能を極限まで活用すべくチューニングされている
  - これを超えるようなものを容易に開発することはできない「金字塔」

© 2021 Susumu Yamazaki

# ハイエンドの組込みシステムに搭載されるSIMD/ベクトル処理機能を活用するには

- BLAS/LAPACKのようなライブラリを活用するときに組込みエンジニアが持つであろう懸念・要望
  - きっとメモリ・ストレージのサイズがかかるだろう
  - 原理や仕組みがわからないと安心して使えない

© 2021 Susumu Yamazaki

# ハイエンドの組込みシステムに搭載されるSIMD/ベクトル処理機能を活用するには

- そこで，本分科会では，次のことをレクチャーします
  - SIMD／ベクトル処理を活用する上で基礎となるSIMD(Single Instruction Multiple Data)の考え方
  - これを生かすためのメモリ配置
  - SIMD命令・ベクトル命令活用の要点
  - BLASのAPIのごく一部を例にして読み取った，SIMD／ベクトル処理プログラミングのヒント
- 逆に扱わないこと
  - BLAS/LAPACKの活用方法について

© 2021 Susumu Yamazaki

私自身はまだまだ研究の途上で達人の域には達していない

今後，見解を変えるかもしれないことをご了承ください

(私の見た「達人」は，もっともっと，すごかったです！)

© 2021 Susumu Yamazaki

# では本題

© 2021 Susumu Yamazaki

# SIMDの考え方

© 2021 Susumu Yamazaki

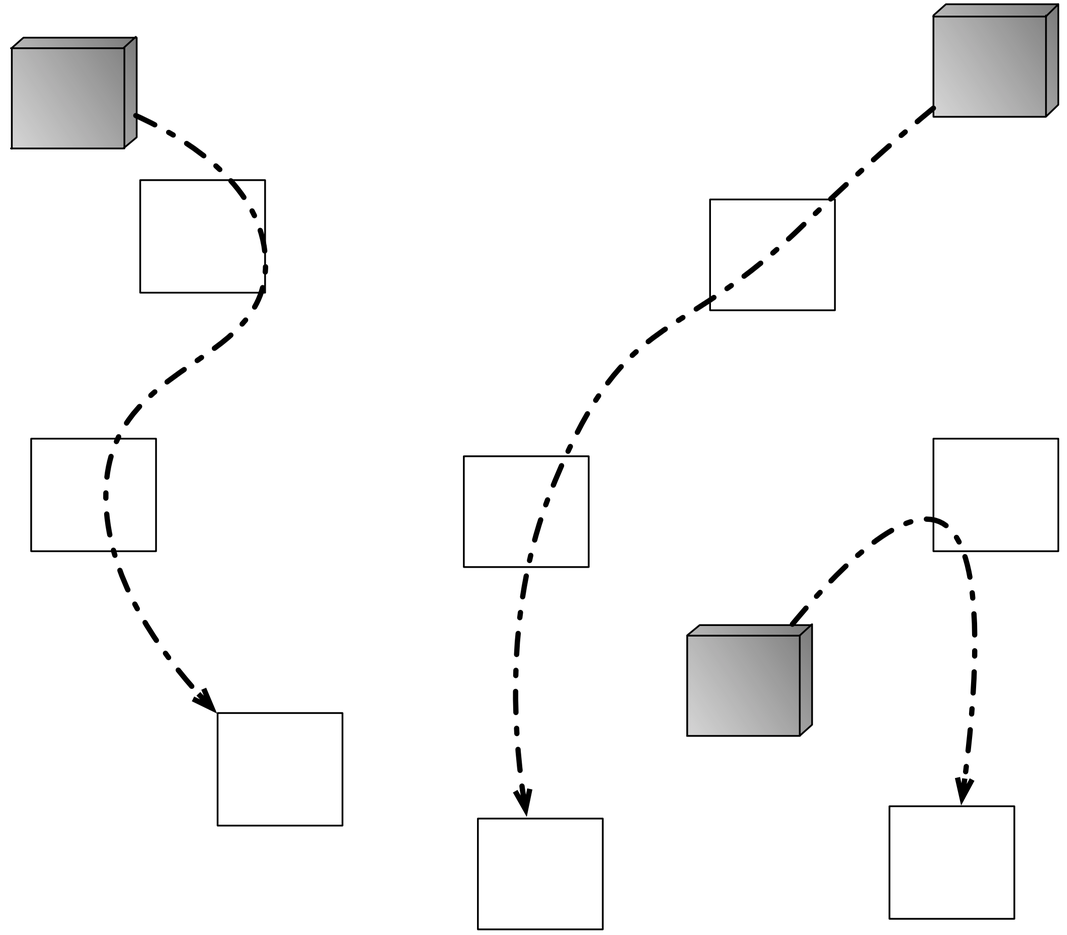
# 並列処理の分類

- SISD(シスディ: Single Instruction Single Data: 単一命令単一データ): 並列ではない
- **SIMD(シムディ: Single Instruction Multiple Data: 単一命令複数データ)**
- MISD(ミスディ: Multiple Instruction Single Data: 複数命令単一データ)
- **MIMD(ミムディ: Multiple Instruction Multiple Data: 複数命令複数データ)**

© 2021 Susumu Yamazaki

# MIMD

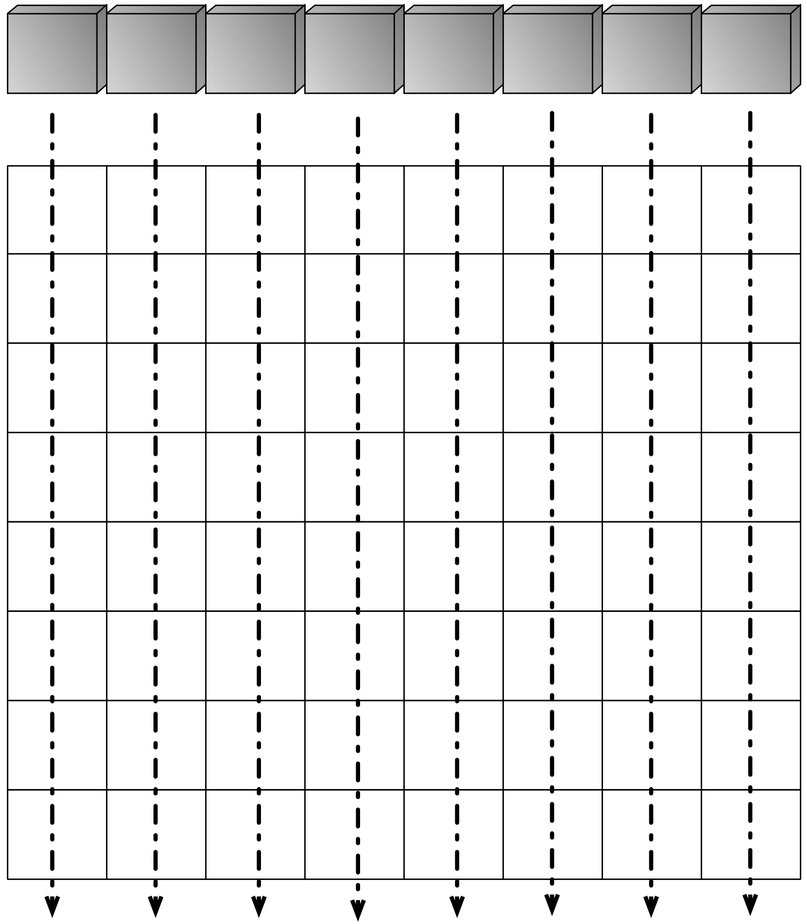
- MIMD (ミムディー，Multiple Instruction Multiple Data): 1つ1つが異なる動作ができる計算ユニットに並列処理させる方式
- たとえるなら，狩をするさいの狩猟犬のように，各々が連携しながら個別の動きをする方式
- マルチコアCPUといったときには，1つ1つのコアが独立して個別の動きをすることができる
- MIMDはSIMDと比べて並列度を稼ぎにくい
  - 市販されているCPUでは数十〜百数十コアがせいぜい
  - 研究段階では千個以上のコアを持つものもある



© 2021 Susumu Yamazaki

# SIMD

- SIMD(シムディ, Single Instruction Multiple Data)
  同じ動作をする計算ユニットに並列処理させる
  方式
- たとえるなら田植えをたくさんの人数で同時に
  行うようなもの
- CPU の SIMD 命令やベクタ命令，GPUで採用
- SIMD方式だと並列度を上げやすく，1000以上
  の並列度を持つGPUが市販されている



© 2021 Susumu Yamazaki

# CPUのSIMD命令

- 固定長のビット幅のSIMDレジスタで演算を行う
  - Intel SSEならば128ビット
  - Intel AVX / AVX2ならば256ビット
  - Intel AVX-512ならば512ビット
  - ARM NEON ならば128ビット(ARMv8の場合)
- SIMDレジスタを区切って使用する
  - 整数ならば{符号付き/なし},{8/16/32/64ビット}
  - 浮動小数点数ならば半精度(16ビット)(ARM NEONのみ)/単精度(32ビット)/倍精度(64ビット)

- 利点: 実現が容易
  - たとえば整数の加減算命令であれば，区切ったビット位置で加算回路のキャリー／ボローを伝播させないようにすれば，実現できる
- 欠点:
  - クロック周波数の向上の場合と異なり，SIMD命令を利用しないと性能は上がらないのでアセンブリプログラミングもしくはSIMD命令のコード生成に対応したコンパイラが必須
  - SIMDレジスタのビット幅が変わると機械語命令の互換性がないので，ビット幅を変える時にプログラミングやコンパイルをしなおす必要がある
- 最近のClangとGCCは auto vectorization が常時有効となっており，**ループに対して自動でSIMD命令を使う**ようにコード生成してくれる

© 2021 Susumu Yamazaki

# ベクトル命令

- ベクトル命令も並列処理の分類だと
  SIMDに区分される
- SIMD命令の欠点:
  - SIMDレジスタのビット幅が変わると
    機械語命令の互換性がないので,
    ビット幅を変える時にプログラミング
    やコンパイルをしなおす必要がある

- これに対しベクトル命令では
  - ベクトルレジスタ幅はCPUによって
    異なるが，それに合わせてループ回数
    などのパラメータを設定する専用命令
    が存在する
  - そのためベクトルレジスタ幅が変わっ
    ても，プログラミングやコンパイルを
    しなおす必要はない

© 2021 Susumu Yamazaki

# ベクトル命令について

© 2021 Susumu Yamazaki

# ベクトル命令について

- RISC-Vのベクトル拡張に対応するCPUを搭載したIoTボードを入手できました🎉
- ベクタ命令をコンパイルするクロスコンパイル環境の構築に成功しました🎉
- ベクタ命令を使ったサンプルアセンブリコードの実行にも成功しました🎉
- ただ，ベンチマーク測定の環境を整えるのが間に合いませんでした😭
- そのため，動作確認以上のことは出来ていません(申し訳ないです)😭

© 2021 Susumu Yamazaki

# そういうわけで

© 2021 Susumu Yamazaki

以降，SIMD命令にフォーカスします

ご了承ください 😞

© 2021 Susumu Yamazaki

# ところで

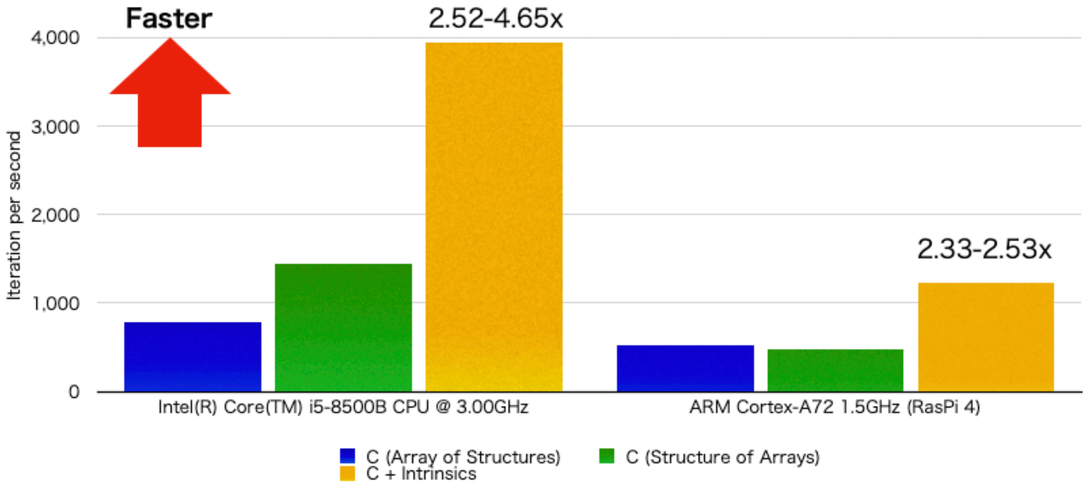
© 2021 Susumu Yamazaki

# SIMD化することで
# どれくらい効果があるのか？

© 2021 Susumu Yamazaki

# サンプルプログラム作ってみました

- https://github.com/zacky1972/simd_sample
- モノクロ化の画像フィルタを題材に
- 8ビット符号なし整数→32ビット符号なし整数→32ビット浮動小数点数
  →{R, G, B} <= 0.299 * r + 0.587 * g + 0.114 * b
  →小数点以下四捨五入→32ビット符号なし整数→8ビット符号なし整数
- C言語で書いた場合 (2種類: 後述)
  Auto-vectorizationのためループでSIMDコードを生成
- Intrinsic (SIMD命令をCで記述する方法の1つ)で書いた場合
- Intel Core i5 / AVX2
- ARMv8 / NEON
- C言語+Auto-vectorizationに比べてSIMD命令を手で記述して最適化を図ると
  2〜5倍の高速化効果がある
- しかもSIMD命令の記述にはまだ最適化の余地がありそうな感触がある




| | C (Array of Structures) | C (Structure of Arrays) | C + Intrinsics |
|---|---|---|---|
| Intel(R) Core(TM) i5-8500B CPU @ 3.00GHz | 781 | 1,441 | 3,933 |
| ARM Cortex-A72 1.5GHz (RasPi 4) | 526 | 482.8 | 1,223 |

© 2021 Susumu Yamazaki

2倍も高速になる 😲

vs

たかだか2倍しか高速にならない 🙅‍♀️

© 2021 Susumu Yamazaki

たかだか2倍程度しか高速にならない

のに

Intrinsics / アセンブリコードのプログラミングをするの？🤔

© 2021 Susumu Yamazaki

後述🤔🤔🤔

© 2021 Susumu Yamazaki

# 閑話休題

© 2021 Susumu Yamazaki

# SIMDを生かすための
# メモリ配置

© 2021 Susumu Yamazaki

# Array of Structures (AoS)
# Structure of Arrays (SoA)

© 2021 Susumu Yamazaki

# Array of Structures (AoS)

- 構造体の配列
- AoSはSIMD化しにくい
  - 飛び飛びに参照する必要がある
  - RGB 8ビットずつだった場合には,
    3バイト飛ばししながらr, g, bをロードして
    モノクロ化の係数をかけてやる必要がある
- ただし NEON にはそのための便利なロード・ストア命令が備わっている！
  - この場合だと3バイトずつ読み飛ばしてSIMDレジスタに格納していくロード命令 ld3q が用意されている

```
for(uint_fast32_t k = 0; k < LOOP; k++) {
    for(uint_fast16_t i = 0; i < SIZE; i++) {
        uint8_t p
          = (uint8_t)round(
              0.299 * pixel[i].r
              + 0.587 * pixel[i].g
              + 0.114 * pixel[i].b
            );
        pixel[i].r = p;
        pixel[i].g = p;
        pixel[i].b = p;
    }
}
```

© 2021 Susumu Yamazaki

# Structure of Arrays (SoA)

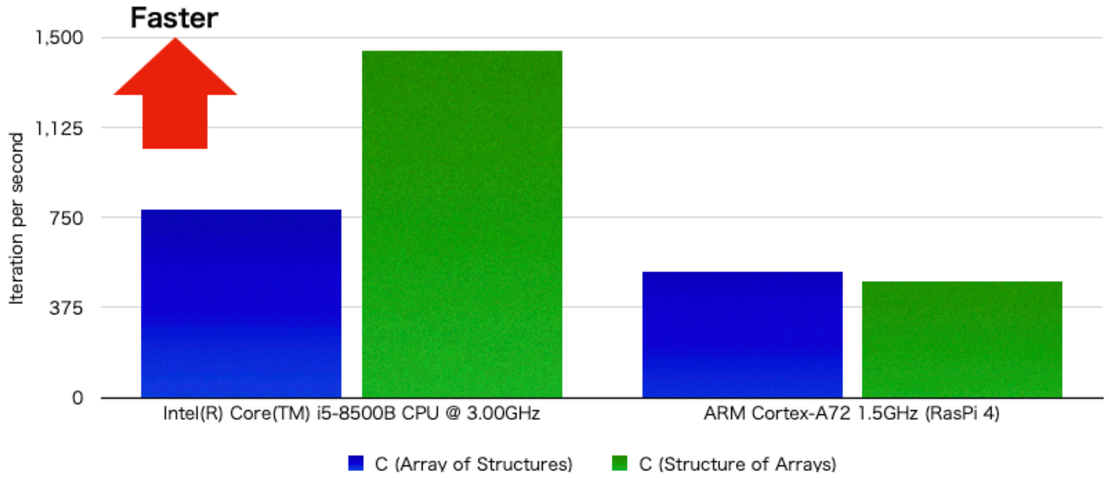
- 配列の構造体
- AoSよりもSoAの方がSIMD化しやすい！
  - 8ビット配列の値を
    SSE・NEONの場合128ビット(16個分)ずつ
    AVX2の場合256ビット(32個分)ずつ
    AVX-512の場合512ビット(64個分)ずつ
    まとめて処理する形にしやすい

```
for(uint_fast32_t k = 0; k < LOOP; k++) {
    for(uint_fast16_t i = 0; i < SIZE; i++) {
        uint8_t p
          = (uint8_t)round(
              0.299 * pixel_r[i]
              + 0.587 * pixel_g[i]
              + 0.114 * pixel_b[i]
            );
        pixel_r[i] = p;
        pixel_g[i] = p;
        pixel_b[i] = p;
    }
}
```

© 2021 Susumu Yamazaki

# ベンチマーク結果をもう一度見てみよう

- https://github.com/zacky1972/simd_sample
- Intel / AVX2 だと AoSよりSoAが2倍近く速い
  - SoAの方がSIMD化しやすいという定石通り
- ARMv8 / NEONだと微妙にAoSの方が速い
  - NEONに用意されている3バイト飛ばしのロード・ストア命令の効果か？
  - 構造体のサイズが5バイト以上になると対応する命令がなくなるので，おそらく遅くなるのでは？




| | C (Array of Structures) | C (Structure of Arrays) |
|---|---|---|
| Intel(R) Core(TM) i5-8500B CPU @ 3.00GHz | 781 | 1,441 |
| ARM Cortex-A72 1.5GHz (RasPi 4) | 526 | 482.8 |

© 2021 Susumu Yamazaki

Auto-vectorizationを使う範囲では
AoSよりもSoAのスタイルに
メモリ配置した方が無難そう

© 2021 Susumu Yamazaki

# SIMD命令・ベクトル命令
# 活用の要点

© 2021 Susumu Yamazaki

# SIMD命令・ベクトル命令活用の要点

- モノクロ化の場合の流れ(これら1つ1つの手順は定石パターンになりそう)
    1. ロード
    2. 8ビットから16ビットに拡張して下位と上位に分割
    3. 16ビットから32ビットに拡張して下位と上位に分割
    4. 32ビット整数から32ビット浮動小数点数に変換
    5. 積→積和→積和
    6. 32ビット浮動小数点数を四捨五入して32ビット整数に変換
    7. 32ビットから16ビットに縮小しつつ下位と上位を統合
    8. 16ビットから8ビットに縮小しつつ下位と上位を統合
    9. ストア

- CPUのSIMDレジスタ数を超過しないようにスケジューリングする
    - 拡張して下位と上位に分割する際に，上位の後続の計算を後回しにする
    - R, Gの処理を先にやって，積→積和してSIMDレジスタをまとめた後 Bの処理を行なって積和する
- 変数に型を明記するとわかりやすい (ハンガリアン記法的なアプローチ)
    - 例:
        - float32x4_t **f32x4**_pixel_r;
        - uint8x16_t **u16x8**_pixel_b;
- NEONは命名規則がわかりやすく，
  複雑なIntel SSE/AVX よりもプログラミングしやすい

© 2021 Susumu Yamazaki

# OpenBLASに見る
# SIMD処理プログラミングのヒント

© 2021 Susumu Yamazaki

# OpenBLAS

- https://github.com/xianyi/OpenBLAS
- BLAS: 基本線形代数サブプログラム
- OpenBLASは最適化されたBLASのライブラリでBSDライセンスで公開されているもの
- 使い勝手よりスピードを追求してAPIが設計されている
- それでいて汎用性があるような絶妙さがある

```
DGEMM  performs one of the matrix-matrix operations

   C := alpha*op( A )*op( B ) + beta*C,

where  op( X ) is one of

   op( X ) = X   or   op( X ) = X**T,

alpha and beta are scalars, and A, B and C are matrices, with op( A )
an m by k matrix,  op( B )  a  k by n matrix and  C an m by n matrix.
```

BLAS (Basic Linear Algebra Subprograms) https://www.netlib.org/blas/

- たとえばGEMM(行列と行列の積)
  - データ型に合わせて最適化されている
    - SGEMM(単精度版)
    - DGEMM(倍精度版)
    - CGEMM(複素数単精度版)
    - ZGEMM(複素数倍精度版)
  - 次の計算を同時に実行することで，大域的に最適化できる
    - 行列の乗算
    - 行列の転置
    - 行列のスカラー倍
    - 行列の加算

© 2021 Susumu Yamazaki

# OpenBLAS

- アセンブリコード例
  - https://github.com/xianyi/OpenBLAS/blob/develop/kernel/arm64/dgemm_kernel_4x4.S
  - https://github.com/xianyi/OpenBLAS/blob/develop/kernel/arm64/dgemm_kernel_4x8.S
  - https://github.com/xianyi/OpenBLAS/blob/develop/kernel/arm64/dgemm_kernel_8x4.S

- 演算カーネルがCPUアーキテクチャやサイズで細かく場合分けされている
- マクロ等を定義することで同じようなコード列を巧みに再利用している
- 内側ループを展開して速度を向上させるようにしている

© 2021 Susumu Yamazaki

# まとめ

© 2021 Susumu Yamazaki

# まとめ

- SIMDは同じ動作をする計算ユニットに並列処理させる方式
- CPUのSIMD命令，ベクトル命令，GPUがSIMDに該当
- CPUのSIMD命令は固定長のビット幅のSIMDレジスタで演算を行う
- SIMDレジスタのビット幅が変わると機械語命令の互換性がない
- SIMD命令のビット幅を変える時にプログラミングやコンパイルをしなおす必要がある
- ベクトル命令ではベクトルレジスタ幅が変わっても，プログラミングやコンパイルをしなおす必要はない
- C言語+Auto-vectorizationに比べてSIMD命令を手で記述して最適化を図ると2〜5倍の高速化効果がある
- Auto-vectorizationを使う範囲ではAoS(構造体の配列)よりもSoA(配列の構造体)のスタイルにメモリ配置した方が無難そう
- 定石パターンをマクロ化すると良さそう
- CPUのSIMD/ベクトルレジスタ数を超過しないようスケジューリングする
- 変数に型を明記するとわかりやすい (ハンガリアン記法的なアプローチ)
- NEONは命名規則がわかりやすく，複雑なIntel SSE/AVX よりもプログラミングしやすい
- 使い勝手よりスピードを追求し，それでいて汎用性があるような絶妙さがある感じでAPIを設計する
- 演算カーネルをCPUアーキテクチャやサイズで細かく場合分けする
- マクロ等を定義することで同じようなコード列を再利用する
- 内側ループを展開して速度を向上させる

© 2021 Susumu Yamazaki

# 置いた話を元に戻して

© 2021 Susumu Yamazaki

たかだか2倍程度しか高速にならない

のに

Intrinsics / アセンブリコードのプログラミングをするの？🤔

© 2021 Susumu Yamazaki

手でSIMD命令を書くのは
本当に性能が要求される部分に
限定すべきである

© 2021 Susumu Yamazaki

どのくらい再利用されるか
どのように再利用されるかを
分析してAPIを設計すべきである

© 2021 Susumu Yamazaki

メンテナンスできる要員を
常に確保し続ける覚悟が必要である

© 2021 Susumu Yamazaki

でも

そんなこと

やってられないですよね？

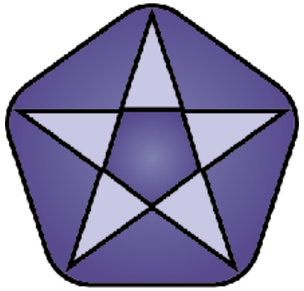
© 2021 Susumu Yamazaki

そこで

並列処理に長けたElixirのプログラムから

超高速なSIMD命令＋マルチコア並列のコードを

生成する技術シーズを育てています🎉

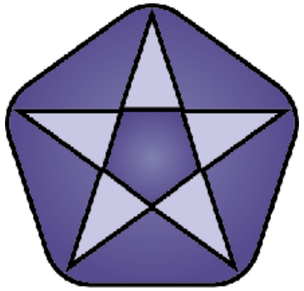


© 2021 Susumu Yamazaki

オンラインの研究会を開催しますので

興味ある人は連絡ください

zacky1972@gmail.com



© 2021 Susumu Yamazaki