


Qiita Engineer Festa 2023の記事投稿キャンペーンに参加してプレゼントを獲得しよう🚀

×

 この記事は最終更新日から1年以上が経過しています。



@benki

投稿日 2022年01月12日 更新日 2022年01月13日 19468 views

Rust のパフォーマンスに何が影響を与えているのか

 Rust



動機

The Rust Performance Book という書きものを見つけました。いろいろなパフォーマンス改善テクニックが書かれているわけですが、実際に普段書いてる Rust コードの中で一体何がパフォーマンスに与える影響が大きいのか？という点が気になってベンチマークを取ってみました。

今回パフォーマンスを計測するプログラムはビットマップ画像（1600px x 1200px）をグレースケールに変換する処理です。I/O のパフォーマンスは無視します。 &[u8] から RGB をそれぞれ 1byte ずつ（合計 3bytes）取ってきて、それをグレースケールの 1byte に変換して Vec<u8> にする時間を計測します。イメージとしては下記のような関数です。

```
// source が カラーのビットマップ画像のデータ
fn sample(source: &[u8]) -> Result<Vec<u8>> {
    let mut v = vec![];
```



48



45



```
// グレースケールに変換して v: Vec<u8> に追加
v.push(to_grayscale_f32(d)?);
}
// グレースケール画像のデータを返す
Ok(v)
}
```

グレースケールへの変換はグレースケール画像のうんちくの CIE XYZ を参考にしました。コードは下記の通りです。

(#[rustfmt::skip] アトリビュートを付けておくと rustfmt で整形されなくなります。可読性のためにわざとインデントを入れたりしていて整形されたくない場合に便利です。)

```
#[rustfmt::skip]
fn to_grayscale_f32(bgr: &[u8]) -> Result<u8> {
    Ok(
        (
            0.0722 * bgr.blue()? as f32 +
            0.7152 * bgr.green()? as f32 +
            0.2126 * bgr.red()? as f32
        )
        as u8
    )
}
```

環境

rustc 1.57.0

Windows 10 64bit

Core i5 8200Y 1.3GHz

メモリアロケーションによる影響

実装

メモリアロケーションが頻繁に発生する状況を想定しています。↓こんなコード書く人はいないと思いますが…。

```
fn lots_of_allocation_f32(source: &[u8]) -> Result<Vec<u8>> {
    let mut v = vec![];
    for d in source.chunks(3) {
        // 無意味に Vec に Vec を追加
        v.push(vec![to_grayscale_f32(d)?]);
    }
    // Vec<Vec<u8>> を Vec<u8> に変換
    let v = v.into_iter().flatten().collect();
    Ok(v)
}
```

比較として `Vec::with_capacity` でメモリアロケーションを抑えた実装を書きました。

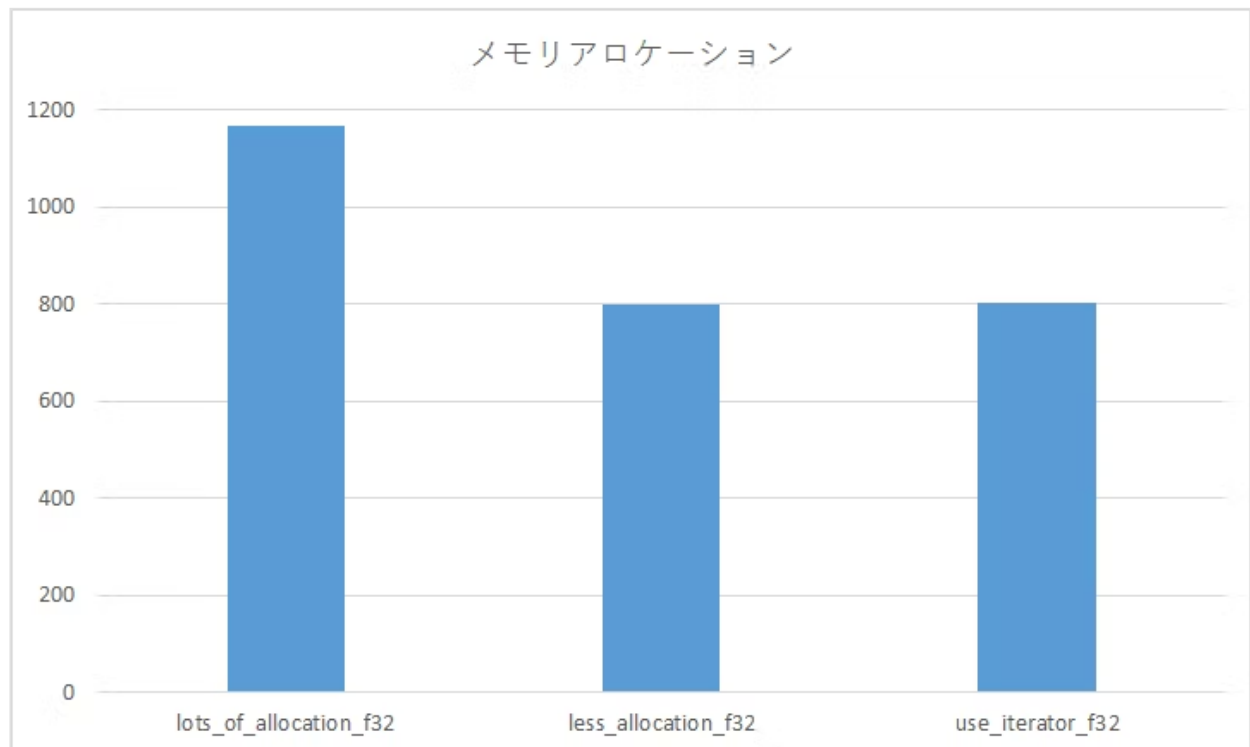
```
fn less_allocation_f32(source: &[u8]) -> Result<Vec<u8>> {
    // あらかじめグレースケール画像サイズ分のメモリ領域を確保しておく
    let mut v = Vec::with_capacity(1600 * 1200);
    for d in source.chunks(3) {
        v.push(to_grayscale_f32(d)?);
    }
    Ok(v)
}
```

もう一つ比較としてイテレータを用いた実装です。すっきり書けて気持ちいい！

```
fn use_iterator_f32(source: &[u8]) -> Result<Vec<u8>> {
    source.chunks(3).map(to_grayscale_f32).collect()
}
```

結果

グラフの縦軸の単位は msec で、関数一回の実行時間です。当たり前ですがメモリアロケーションが頻繁に発生する状況では遅くなります。それにしても $1600 \times 1200 \times 3\text{bytes} \approx 6\text{MB}$ のビットマップ画像をグレースケールに変換する処理にしては全体的に遅すぎますね。メモリアロケーション以外にも何か良くないことが起こっていそうです。(もちろん `--release` オプション付きです。) また、イテレータを使ってもパフォーマンスに差はでないのですね。以後のパフォーマンス比較ではイテレータを使うことにします。



遅延評価による影響

[リンク先](#)にも書いてある通り、`Option::ok_or` はエラーではないときにも `ok_or` の中が評価されます。なので、関数が返す `Option` を `ok_or` で `Result` に変換するときは `ok_or` の中で高コストな処理はしてはいけません。もしくは `ok_or_else` を使ってエラーのときだけ処理が実行されるようにする必要があります。

実装

RGB の 3bytes [u8; 3] から 1byte ずつ抜き出す処理は下記のように実装していました。エラーではないときも anyhow! マクロが評価されてしまいます。

```
use anyhow::{anyhow, Result};

impl GetByte for &[u8] {
    fn byte(&self, index: usize) -> Result<u8> {
        self.get(index).copied().ok_or(anyhow!("error"))
    }
}
```

遅延評価されるように ok_or_else に変えてみます。

```
use anyhow::{anyhow, Result};

impl GetByte for &[u8] {
    fn byte(&self, index: usize) -> Result<u8> {
        self.get(index).copied().ok_or(anyhow!("error"))
    }
    fn byte_lazy(&self, index: usize) -> Result<u8> {
        self.get(index).copied().ok_or_else(|| anyhow!("error"))
    }
}
```

さらに anyhow::Context を使った場合と、unsafe な実装もパフォーマンス比較用に準備しました。

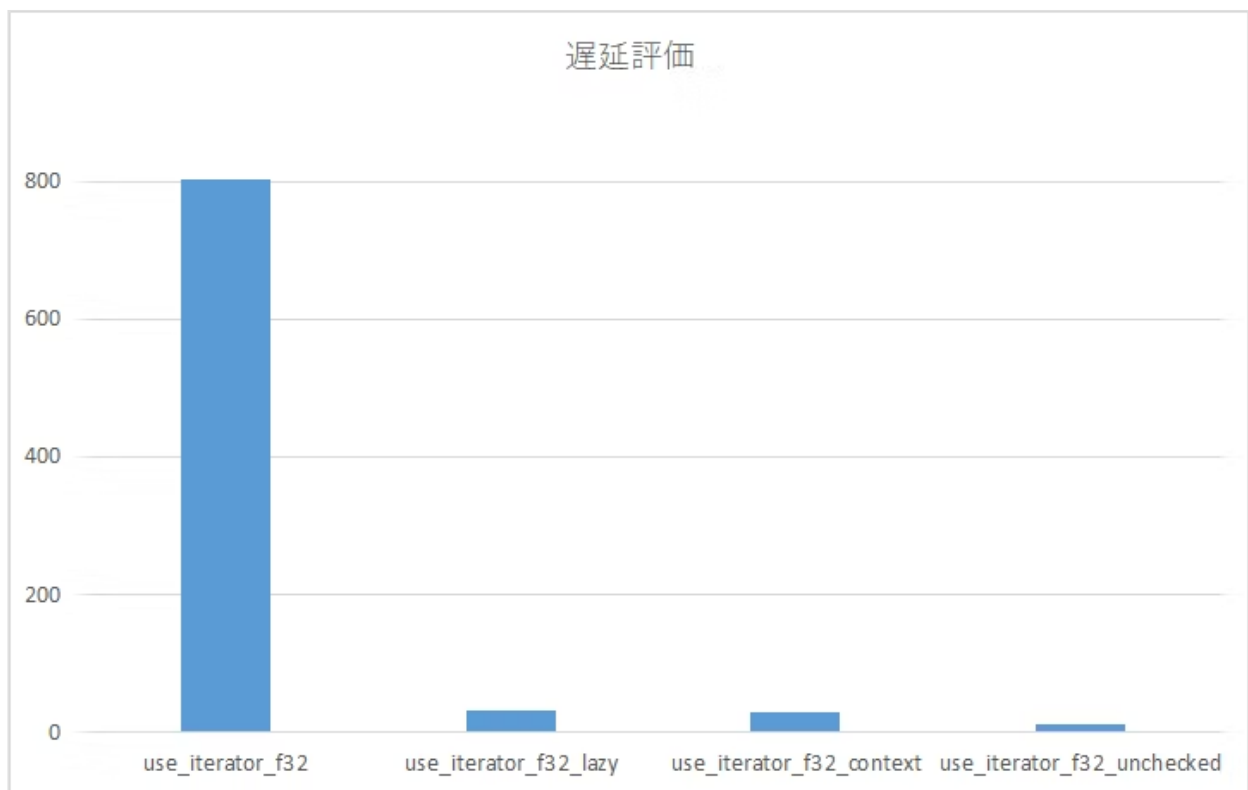
```
use anyhow::{anyhow, Context, Result};

impl GetByte for &[u8] {
    fn byte(&self, index: usize) -> Result<u8> {
        self.get(index).copied().ok_or(anyhow!("error"))
    }
    fn byte_lazy(&self, index: usize) -> Result<u8> {
        self.get(index).copied().ok_or_else(|| anyhow!("error"))
    }
    fn byte_context(&self, index: usize) -> Result<u8> {
        self.get(index).copied().context("error")
    }
}
```

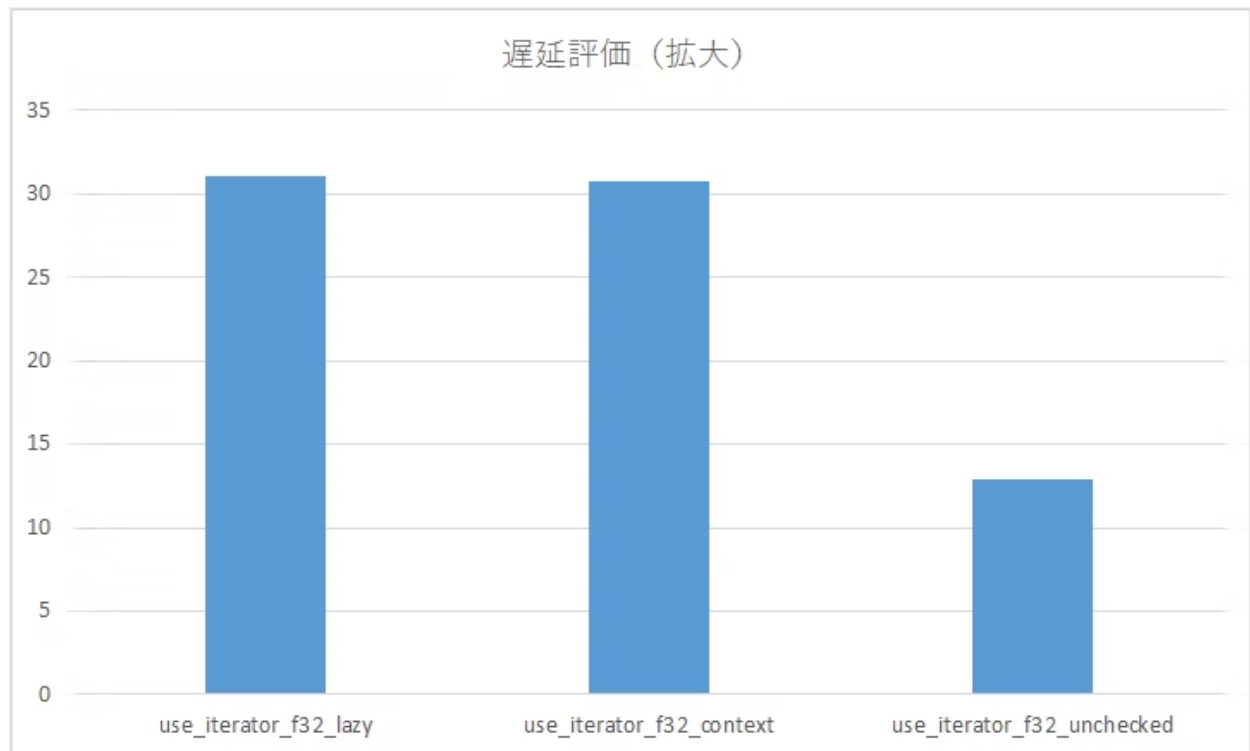
```
unsafe { *self.get_unchecked(index) }  
}  
}
```

結果

左から順に、遅延評価しない場合、遅延評価した場合、`anyhow::Context` の場合、エラーチェックしない場合です。圧倒的な差です。遅延評価するようにしましょう。
(`cargo clippy` すると遅延評価するように教えてくれます。)



右三本が見えないので拡大しました。`unsafe` パワーは魅力的ですが、ちゃんとエラーチェックはしましょう。



浮動小数点数による影響

浮動小数点数演算は幾分かコストが掛かります。これは Rust に限った話ではなく、他の言語でも同じですね。

グレースケール変換で浮動小数点数 f32 を使っていますが、整数 u16 で変換するよう書き換えてみます。

実装

整数演算してからビット演算 $\gg 8$ でもって 256 で割っています。

```
// 浮動小数点数演算
#[rustfmt::skip]
fn to_grayscale_f32(bgr: &[u8]) -> Result<u8> {
    Ok(
        (
```

```

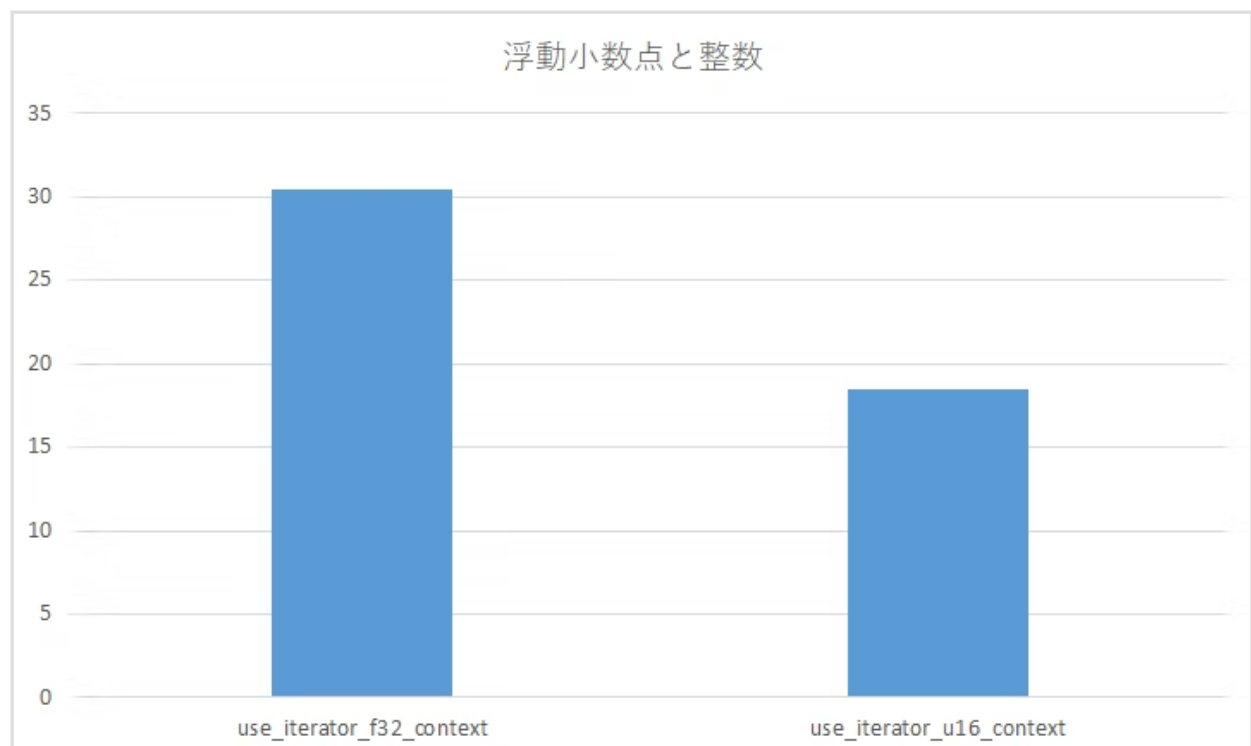
        0.2126 * bgr.red()? as f32
    )
    as u8
)
}

// 整数演算
#[rustfmt::skip]
fn to_grayscale_u16(bgr: &[u8]) -> Result<u8> {
    Ok(
        (
            ((19 * bgr.blue()? as u16) >> 8) +
            ((183 * bgr.green()? as u16) >> 8) +
            ((54 * bgr.red()? as u16) >> 8)
        )
        as u8
    )
}

```

結果

なるべく整数演算しよう。



ビルドオプションによる違い

実装

--release をオプション付けただけのビルドと、さらに panic="abort" を設定したときと、lto="fat" を設定したときのパフォーマンスを比較します。

Cargo.toml

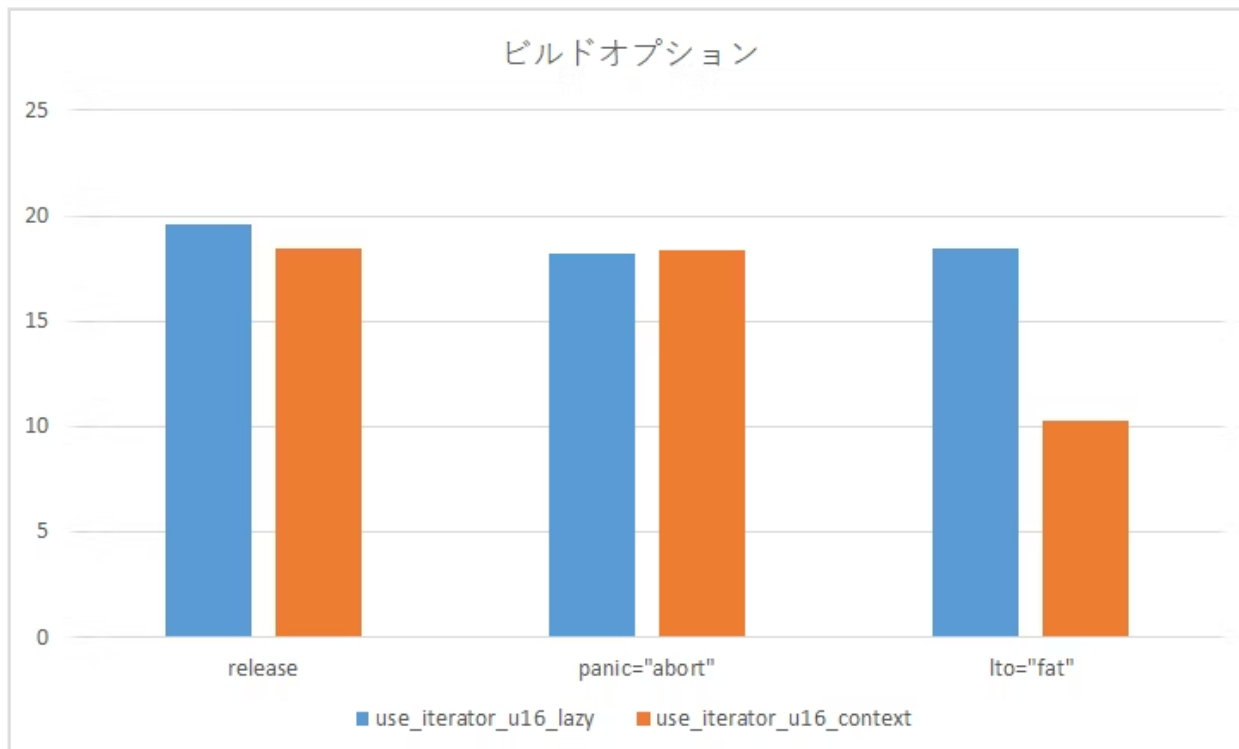
```
[profile.release]
panic = "abort"
```

Cargo.toml

```
[profile.release]
lto = "fat"
codegen-units = 1
panic = "abort"
```

結果

anyhow::Context を使って、lto="fat" を設定した場合が異常に早いです。何かアグレッシブな最適化が働いているのでしょうか？謎です。コンパイル時間は長くなりますが、lto="fat" を設定しよう。



まとめ

❌ `Option::ok_or` の中に高コストな処理を書いてはいけない。

今回は [The Rust Performance Book](#) 中のほんの一部ですが、パフォーマンスに与える影響を調べてみました。他にもいろいろなテクニックが書いてあるので目を通しておくとよいでしょう。おわり。



コメント

この記事にコメントはありません。

編集

プレビュー



テキストを入力

 画像を選択

0B / 100MB

投稿する

How developers code is here.

© 2011-2023 Qiita Inc.

ガイドとヘルプ

About

利用規約

プライバシーポリシー

ガイドライン

デザインガイドライン

ご意見

ヘルプ

広告掲載

コンテンツ

リリースノート

公式イベント

公式コラム

募集

アドベントカレンダー

Qiita 表彰プログラム

API

SNS

Qiita（キータ）公式

Qiita マイルストーン

Qiita 人気の投稿

Qiita（キータ）公式



48

45

Qiita 関連サービス

Qiita Team

Qiita Jobs

Qiita Zine

Qiita 公式ショップ

運営

運営会社

採用情報

Qiita Blog

