

---

知能制御システム学

# 画像処理の高速化

東北大学 大学院情報科学研究科

鏡 慎吾

swk(at)ic.is.tohoku.ac.jp

2008.07.22

# 今日の内容

ビジュアルサーボのようなリアルタイム応用を考える場合、  
画像処理を高速に実装することも重要となる

いくつかの基本的な知識を押さえておかないと、同じアルゴリズムを実行しているのに性能が上がらないということがしばしば生じる

今日は、あくまで普通のPC上での「プログラムの書き方」のレベルで、高速化を考慮する際に知っておくべき基本的な考え方を学ぶ

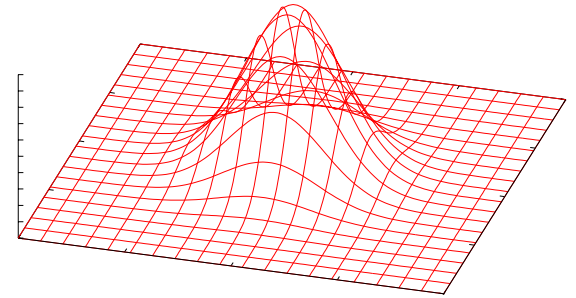
# リアルタイム処理と高速化

- 「リアルタイム」=「高速」ではない
- 目標となる時間制約が定められているのがリアルタイム処理である
  - 34 ms → 33 ms
  - 33 ms → 32 ms
  - どちらも 1 ms の短縮だが、例えばビデオレートでのリアルタイム処理が目的なのであれば、両者の意味は大きく違う
- 平均実行時間 ←→ 最悪実行時間
- 計算時間を短くするには計算量を減らすのが基本。 **だがそれだけではない**

# 例: ガウス平滑化

- $m \times n$  のフィルタマスクによるガウス平滑化は, 単純に実行すると 1 画素当たり  $O(mn)$  の計算量が必要

$$G_{x,y} = \sum_i \sum_j w_{i,j} F_{m+i,n+j}$$
$$w_{x,y} = \frac{1}{2\pi\sigma^2} \exp\left\{-\frac{x^2 + y^2}{2\sigma^2}\right\}$$



- X方向・Y方向それぞれ1次元ガウス平滑化に分解できることに気づけば, 1画素当たり  $O(\max(m,n))$  の計算量で実行可能

$$w_{x,y} = \frac{1}{\sqrt{2\pi}\sigma} \exp\left\{-\frac{x^2}{2\sigma^2}\right\} \cdot \frac{1}{\sqrt{2\pi}\sigma} \exp\left\{-\frac{y^2}{2\sigma^2}\right\}$$

- このような性質を持つフィルタを「分離可能 (separable)」という

# 実行例

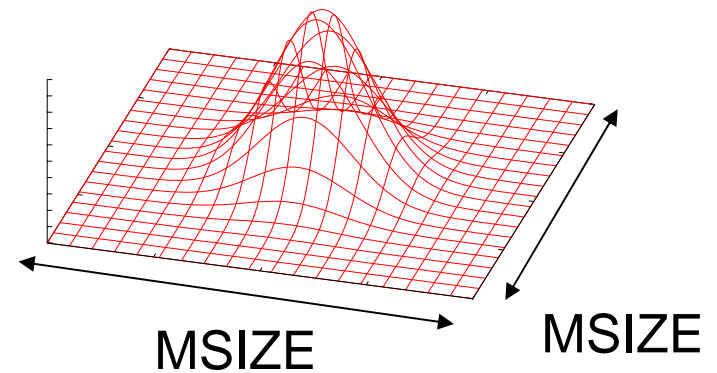
- time\_gauss.cpp  
(時間計測は PerformanceCounter を用いている. ソースコードは perftimer.cpp)

```
// #define OPT_NOSEP  
#define OPT_SEP
```

で, 分離型, 非分離型を切り替え

```
#define MARGIN 7
```

でフィルタマスクのサイズを指定



$$\text{MSIZE} = 2 * \text{MARGIN} + 1$$

# わかったこと

- 計算量が減ったからといって、そのまま計算時間の減少につながるわけではない
- それどころか、条件によっては遅くなる場合すらある

以下で、もう少しシンプルな画像処理について、計算時間に影響を及ぼしそうな代表的な例を見ていく

# サンプルプログラム

- `time_framesub.cpp`  
(同様に `perftimer.cpp` が必要)

簡単なフレーム間差分(が一定量以上の画素を緑色に表示)

```
#define OPT_BASELINE
```

… OpenCVの画像処理関数を(一部)使用した実装

```
#define OPT_NAIVE
```

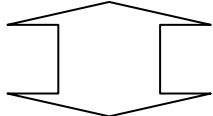
… OpenCVの画像処理関数を使用せずに, 素朴に書いた実装. `baseline` に比べてだいぶ遅い. これを基本として手を加えていく

# 例: ループ交換

```
#define OPT_INTERCHANGE
```

… まず基本中の基本, メモリアクセスのパターンによって速度が変わる例を示す

```
for (j = 0; j < height; j++) {  
    for (i = 0; i < width; i++) {  
        PIXVAL(img, i, j) = ...  
    }  
}
```

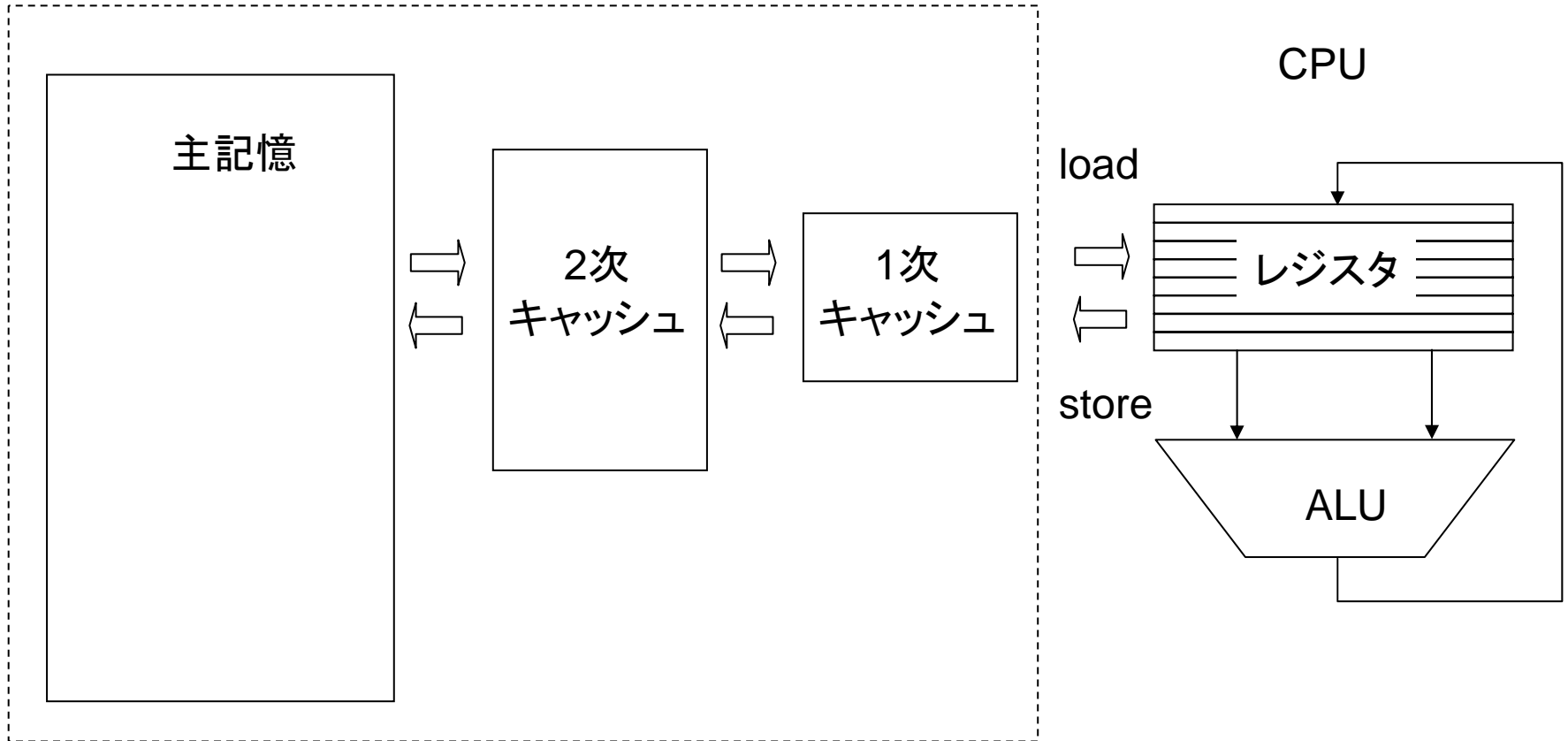
A diagram illustrating row-major traversal. It shows a rectangle with a horizontal line across the middle. On the left side, there are two vertical lines extending from the top and bottom edges to the horizontal line, representing the start of two rows. On the right side, there are two vertical lines extending from the top and bottom edges to the horizontal line, representing the end of two rows. This visualizes the path of a loop that iterates over each row from top to bottom, and within each row, iterates over each column from left to right.

```
for (i = 0; i < width; i++) {  
    for (j = 0; j < height; j++) {  
        PIXVAL(img, i, j) = ...  
    }  
}
```



# 記憶階層

## メモリシステム



- プログラム(プログラマ)から見た場合, キャッシュの存在は見えない  
(制御が自動的に行われる)

# コンピュータプログラムに関する経験則

## 空間的局所性

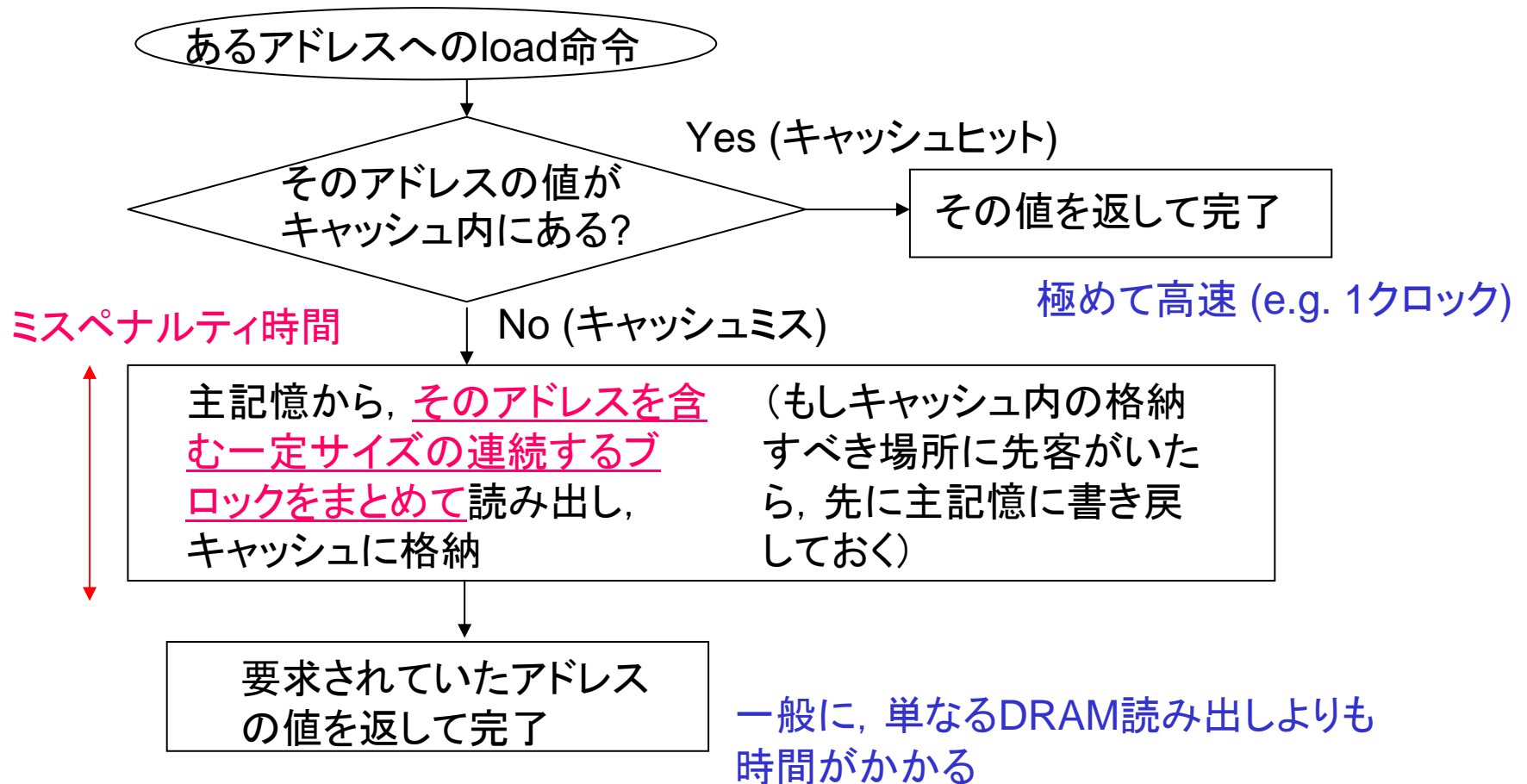
あるデータがアクセスされた場合、その周囲の値もアクセスされる可能性が高い

## 時間的局所性

あるデータがアクセスされる場合、近いうちにその同じデータが再度アクセスされる可能性が高い

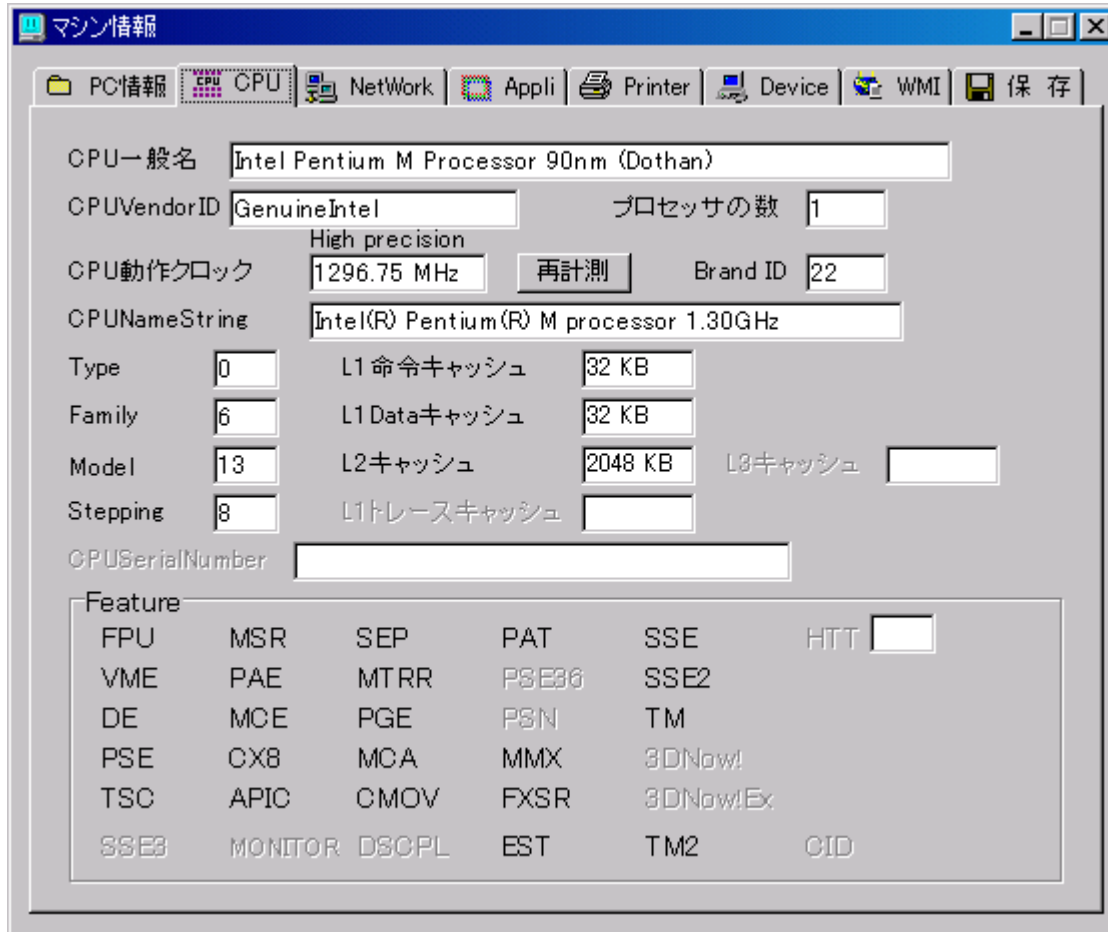
このような経験則を生かすように記憶階層は設計されている。逆に、このような経験則が当てはまらないようなプログラムは性能が出にくい

# キャッシュメモリの動作例



$$\text{平均メモリアクセス時間} = \text{ヒット時間} + \text{キャッシュミス率} \times \text{ミスペナルティ時間}$$

# このPCの場合



2次キャッシュは 2 MB

Dothan の 2次キャッシュ  
は

- 8-way set associative
- 64-byte cache line

PCView, <http://homepage2.nifty.com/smallroom/>

## 64-byte cache line (cache block)

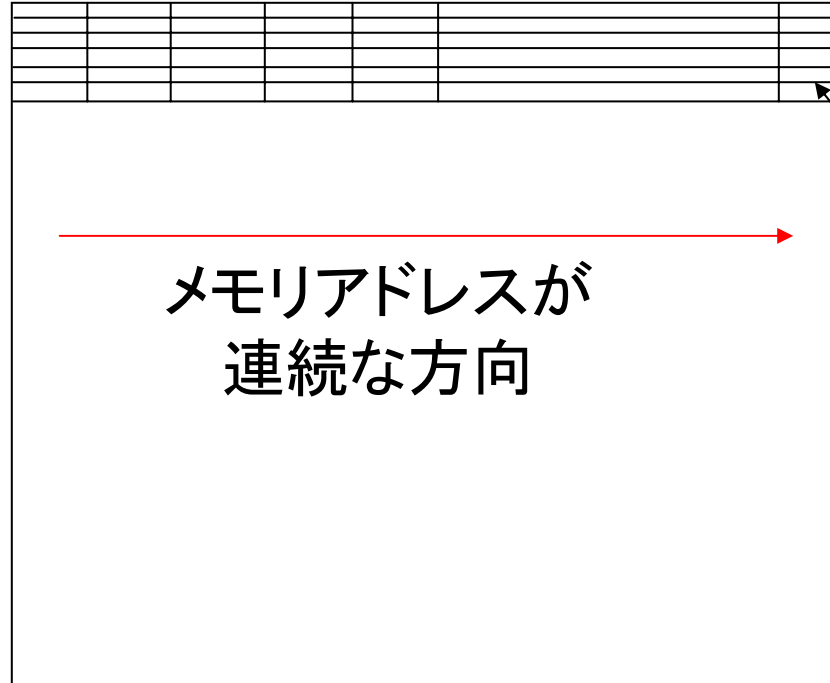
- あるアドレスにアクセスすると, そのアドレスを含む連続する 64 バイトがまとめてキャッシュされる

## 8-way set associative

- ある「64バイトのライン」は, キャッシュ内のどこにでも置けるわけではなく, 8箇所のうちどこかに限定される
- つまり満員でなくてもキャッシュから追い出される場合がある

640

480



640x480画素, 8ビットモノクロ画像が1枚で約 300 KB

連続する64画素が  
キャッシュ1ライン

- 可能な限りメモリアドレスに対して連続にアクセスする方がよい
- キャッシュ内のデータの配置には制約があるので, キャッシュ容量ギリギリまで画像を読み込んでおけるわけではない

# 例: 表引き (table lookup)

事前計算の結果をメモリに保存しておき, 実行時に参照する

```
#define OPT_LUT
```

... 色変換の計算を表引きに置き換える

メモリがボトルネックになり, 速度を稼げない場合もある

# 例: データ転送・コピーの効率化

```
#define OPT_MEMCPY
```

… 現フレームの保存を for ループで1画素ずつコピーする代わりに memcpy() 関数で行う

可能な場合は memcpy(), memset() などを用いる方が高速になることが多い

(訂正) 講義時に, memcpy() はCの標準関数ではないと口走ってしまいましたが, そんなことはなく, しっかりと標準でした



# 例: 画素アドレス計算の最適化

```
#define OPT_INDEX
```

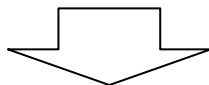
… 画素データが格納されているアドレスを毎回計算するのは(特にシーケンシャルにアクセスする場合は)無駄であり, 可読性を犠牲にすればより高速化可能

```

#define PIXVAL(img, i, j)
(*(uchar*)((img)->imageData + (y) * (img)->widthStep + (x)))

for (j = 0; j < height; j++) {
    for (i = 0; i < width; i++) {
        PIXVAL(img, i, j) = ...
    }
}

```



```

uchar *ptr;
int gap = img->widthStep - width;

for (j = 0, ptr = img->imageData; j < height; j++, ptr += gap) {
    for (i = 0; i < width; i++, ptr++) {
        *ptr = ...
    }
}

```

# 例: ループ融合

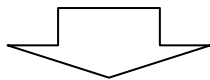
```
#define OPT_FUSION
```

... 複数のループを, ひとつのループに融合

条件分岐のオーバーヘッドが減る効果もあるが, メモリアクセスの局所化による貢献も大きい

逆に, ループ内が独立した複数の処理で構成されているは, 複数のループに分解する方が高速化される場合もある (ループ分散)

```
for (j = 0; j < img->height; j++) {
    for (i = 0; i < img->width; i++) {
        f(PIXVAL(img, i, j));
    }
}
for (j = 0; j < img->height; j++) {
    for (i = 0; i < img->width; i++) {
        g(PIXVAL(img, i, j));
    }
}
```



```
for (j = 0; j < img->height; j++) {
    for (i = 0; i < img->width; i++) {
        f(PIXVAL(img, i, j));
        g(PIXVAL(img, i, j));
    }
}
```

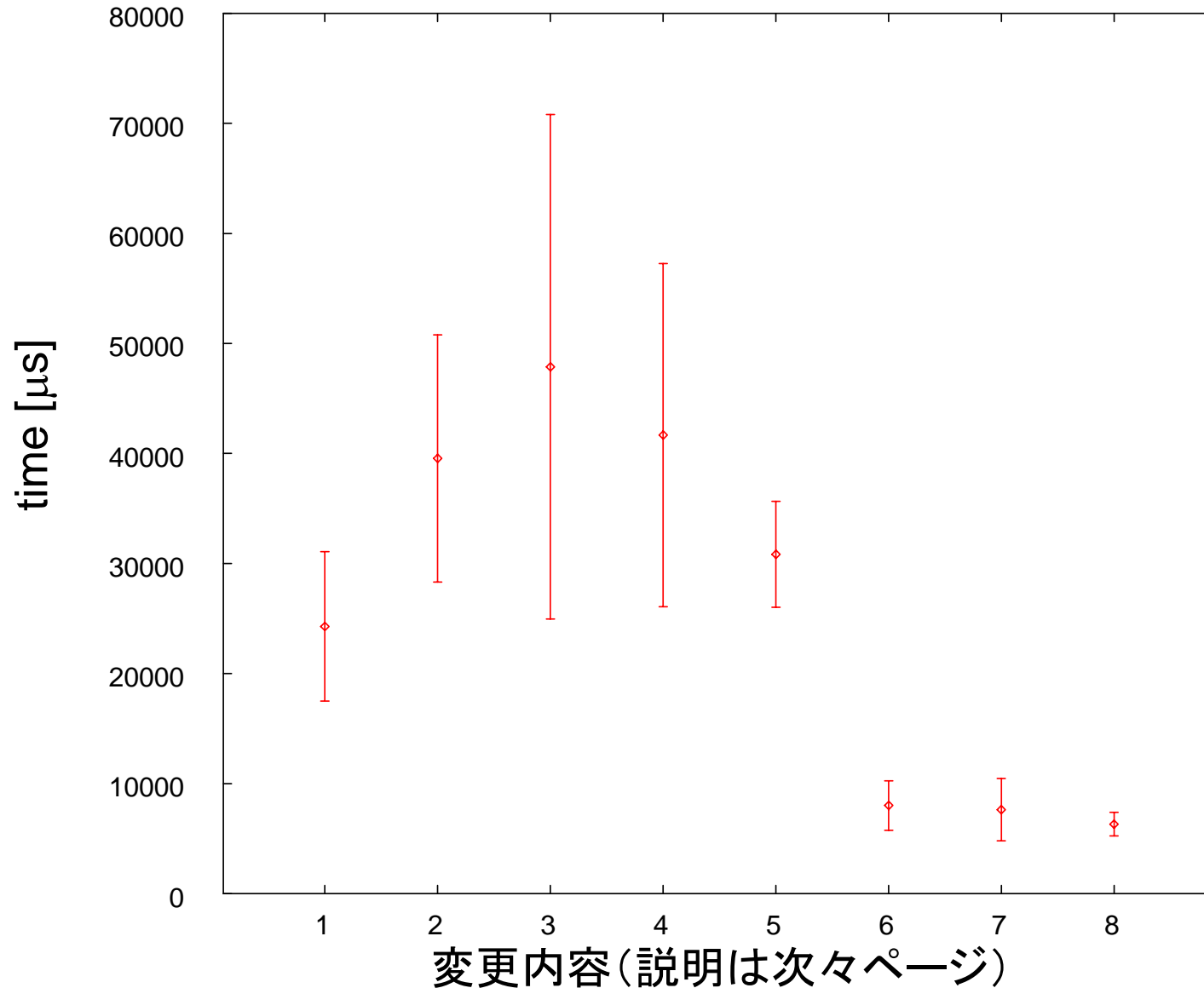
# 例: ループ展開

```
#define OPT_UNROLL
```

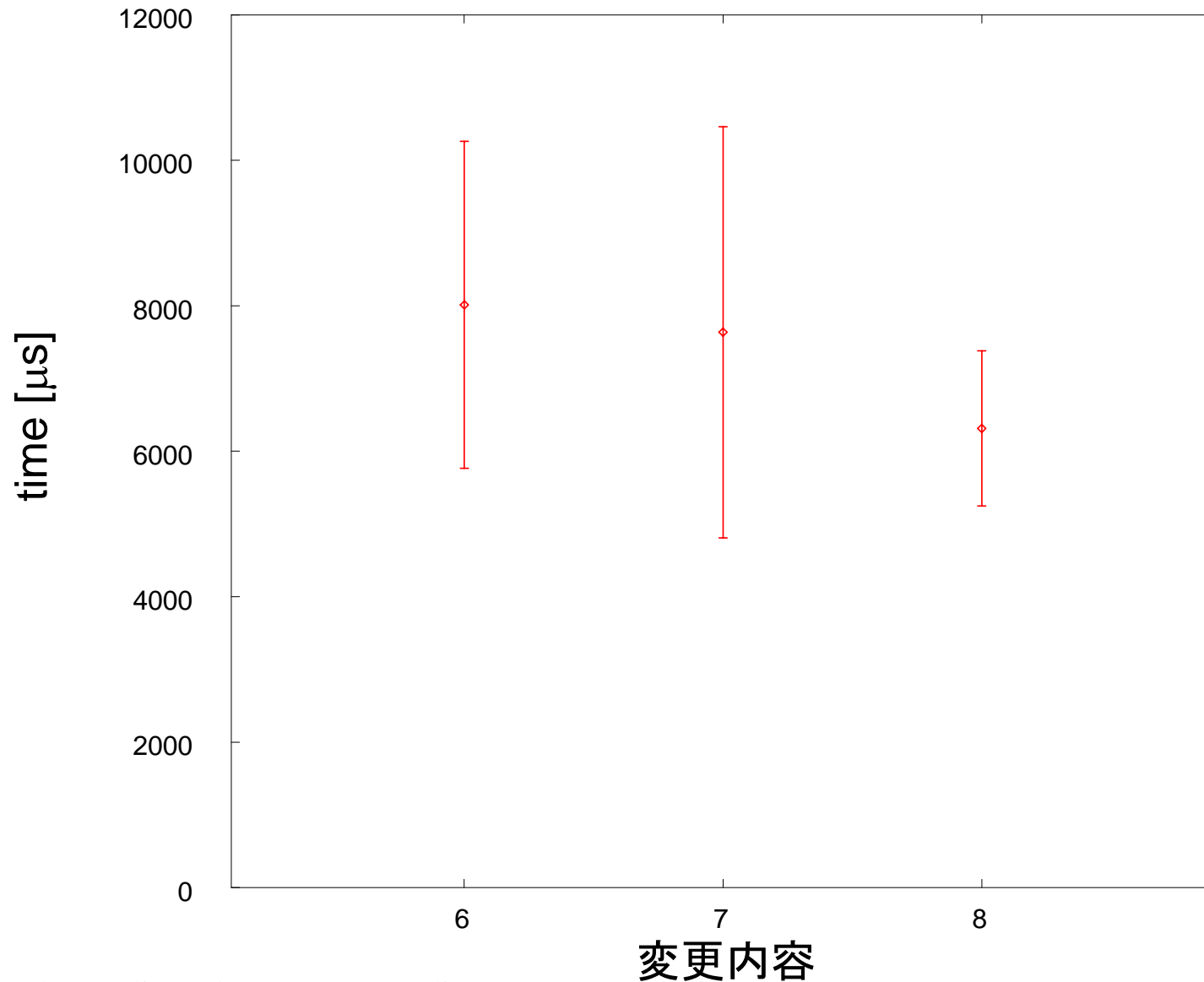
… 繰り返し回数が多いループを, 一部手動で展開する. (例えば1000回のループを100回のループに置きかえ, 1ループ内に10回分の処理を手書きする)

分岐命令のオーバヘッドの削減と, 1ループ内での命令スケジューリングの自由度向上に貢献する

# 結果



# 結果(6, 7, 8 の拡大)



# 各変更内容の説明

## 1. baseline

OpenCV 標準関数を使用

## 2. naive

OpenCV 標準関数を使用せず, 素朴な実装

## 3. interchange

ループ交換により, Y方向にスキャン

## 4. lut

色変換で表引きを行う

## 5. memcpy

表引き + コピー時に memcpy() を使う

## 6. index

表引き + memcpy() + ループインデックスの最適化

## 7. fusion

表引き + memcpy() + ループインデックスの最適化 + ループ融合

## 8. unroll

表引き + memcpy() + ループインデックスの最適化 + ループ融合 + ループ展開



# まとめ

- まず、以上の結果はあくまでも、今回の処理に対して、今回のハードウェアでは結果的にこうだった、というだけである点に注意する
- 今回の条件では、画素アクセスの際のアドレス計算を工夫することで大きな改善が見られた
- 一般に、計算の高速化とコードの保守性・可読性を両立するのは簡単ではないため、バランス感覚が重要
- リアルタイム用途の場合は、平均時間だけでなく最悪時間にも注意する

# さらなる高速化のために

ハードウェア拡張を利用することも、特に今後は重要になると思われる

- SIMD命令 (MMX, SSE, SSE2): 通常はアセンブラレベルのプログラミングが必要
- グラフィックプロセッサ (GPU) の利用: 高級言語でプログラミングできる環境が利用できる場合が多く、注目されている