知能制御システム学

画像処理の高速化
— OpenCV による基礎的な例 —

東北大学 大学院情報科学研究科
鏡 慎吾
swk(at)ic.is.tohoku.ac.jp

2007.07.03

リアルタイム処理と高速化

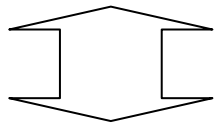
- 「リアルタイム」=「高速」ではない
- 目標となる時間制約が定められているのがリアルタイム処理である．34 ms かかった処理が 33 ms に縮んだだけでも，それによって与えられた時間制約が満たされるのであれば，有用な結果だといえる．
- 「速いアルゴリズムを使う」のがもちろん基本
  e.g. 離散フーリエ変換 → 高速フーリエ変換
- ここでは，アルゴリズムは同じなのに，プログラムの書き方によって速度が変わる典型的な例を見ていく
- 試行錯誤の過程でもある

例: 2重ループの走査順

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. (サンプル: time_framesub.cpp)

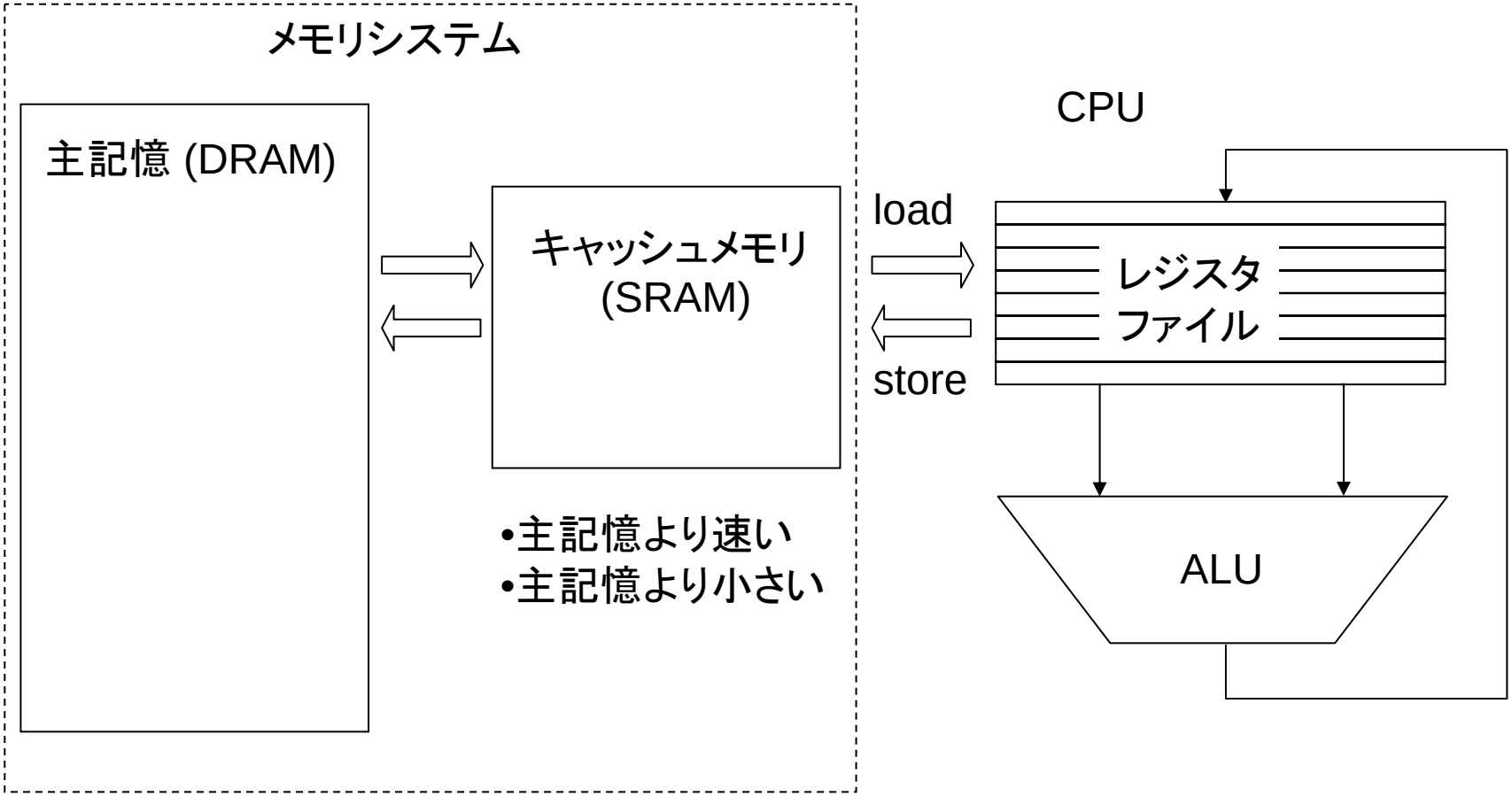
```
for (j = 0; j < img->height; j++) {
    for (i = 0; i < img->width; i++) {
        PIXVAL(img, i, j) = ...

    }
}
```



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

キャッシュメモリ



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

コンピュータプログラムに関する経験則

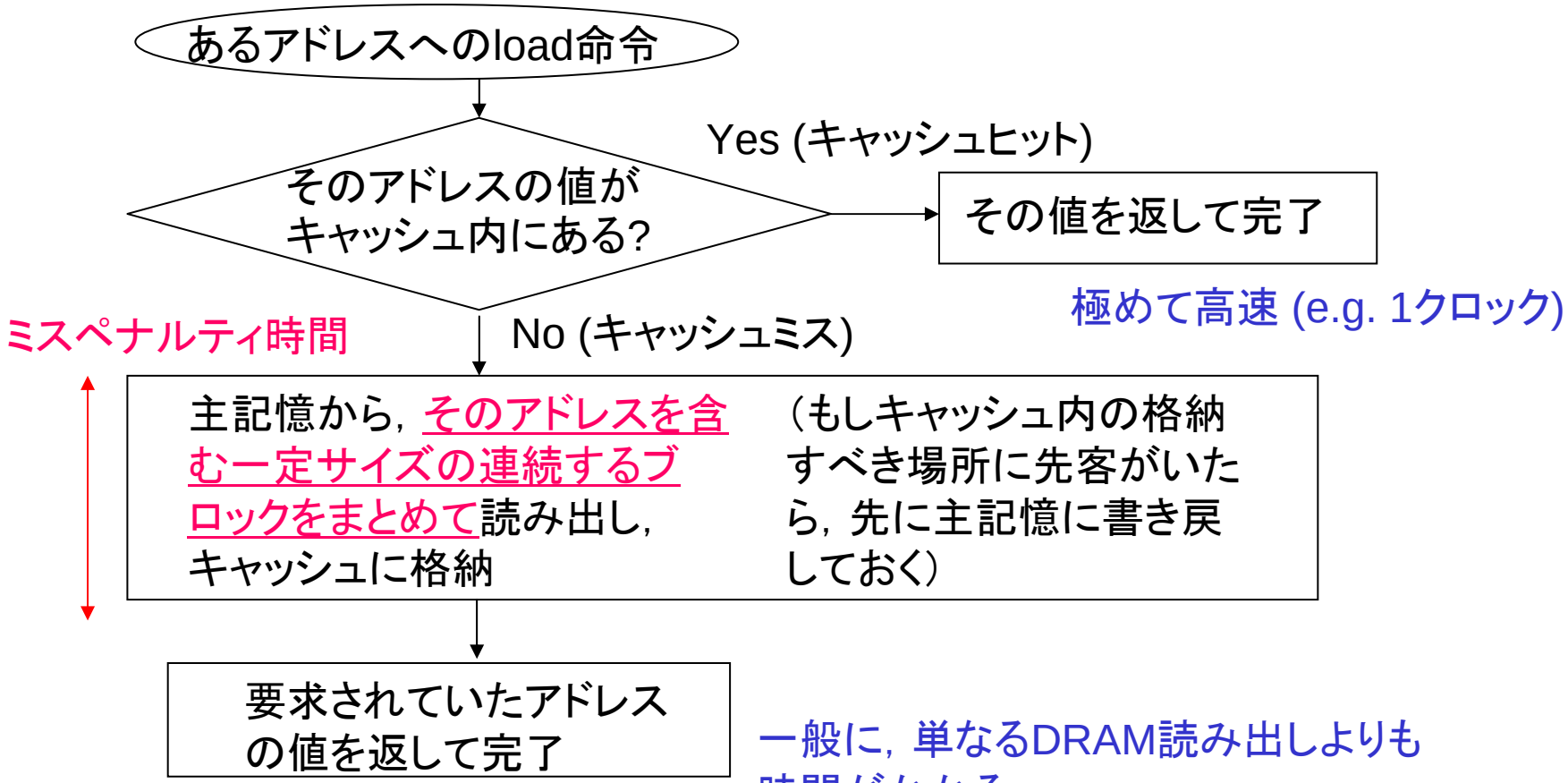
空間的局所性

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

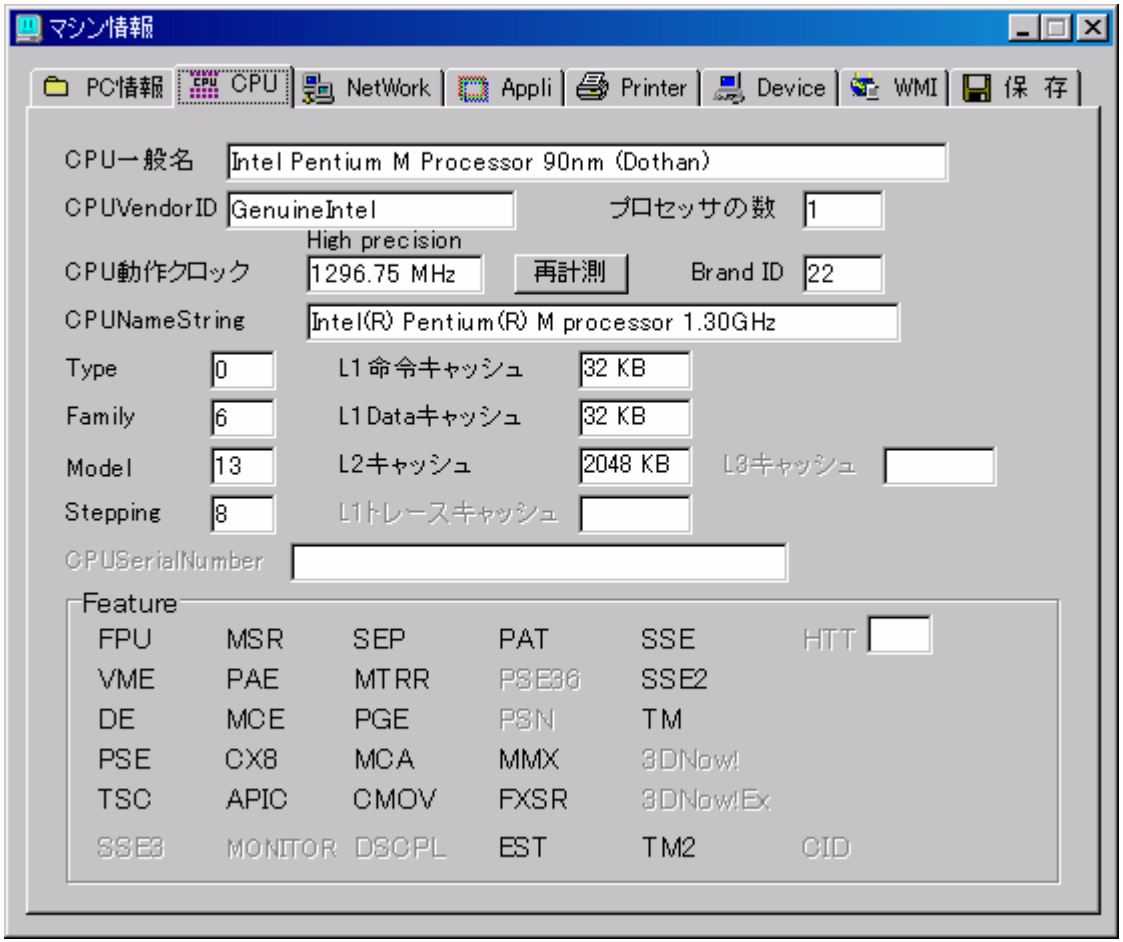
あるデータがアクセスされる場合，近いうちにその同じデータが再度アクセスされる可能性が高い

キャッシュメモリの動作例



平均メモリアクセス時間 = ヒット時間 ＋ キャッシュミス率 × 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

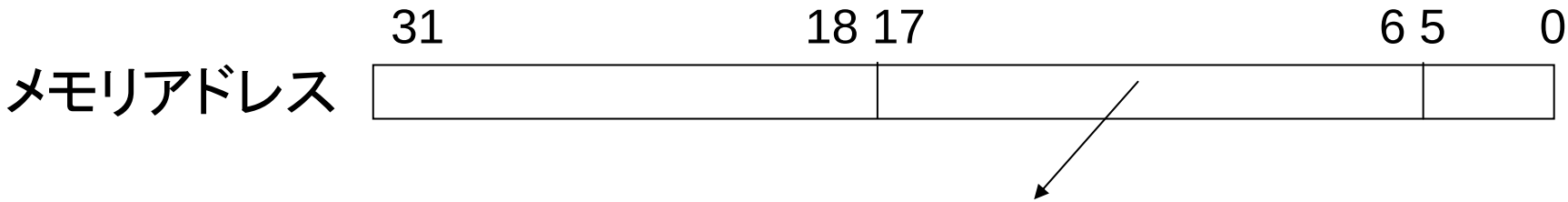
PCView, http://homepage2.nifty.com/smallroom/

64-byte cache line (cache block)

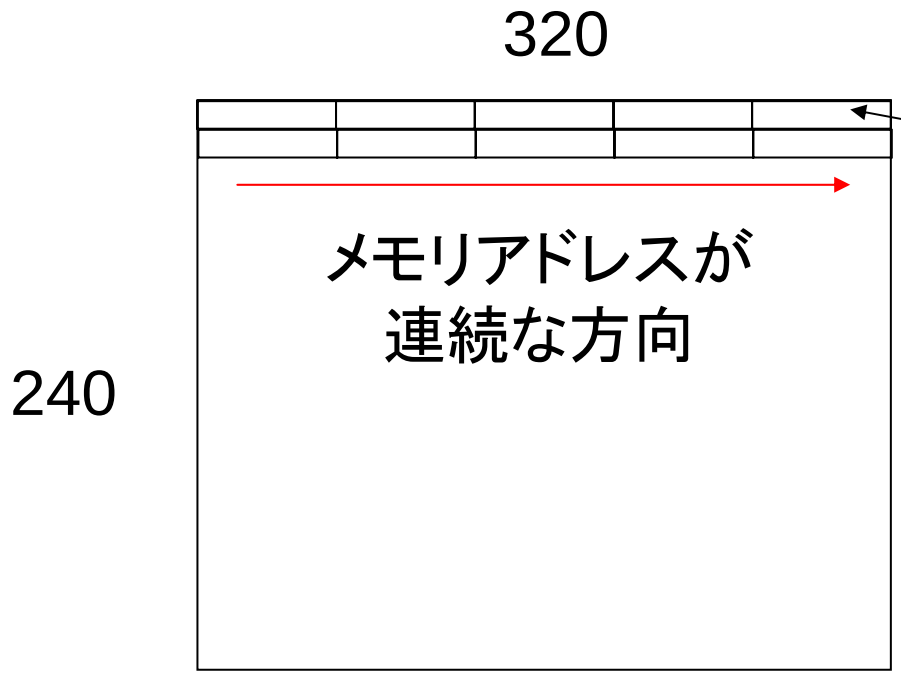
- あるアドレスにアクセスすると，そのアドレスを含む連続する 64 バイトがまとめてキャッシュされる

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



ここの12ビットで，キャッシュ内の位置 (8候補) が決まる

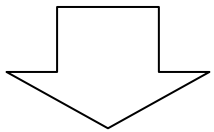


今回のデモで使っている
USBカメラのサイズは
320x240. 8ビット画像であれ
ば，1枚で約 80 KB.

- 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: 処理を局所化する

```
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));
    }
}
```



複数のループを，ひとつのループに統合する方が高速になりやすい

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

- メモリは遅いので，一度（キャッシュに，あるいはレジスタに）読み込んだ値は，可能な限りその場で使い倒すのがよい
- つまり，画像を走査するループをできるだけ少なくまとめ，走査の数を減らすことに相当する
- サンプル: time_framesub_unify.cpp

できるだけ標準関数を使おう

- 自前で書いたルーチン同士で比べたら，走査をまとめた方が断然速いのだが，OpenCV が標準で用意している関数を単純に並べたもの（つまり走査は複数回のまま）よりはむしろ遅くなってしまった

- OpenCV に標準で用意されている関数は十分に高速化されていることが多いので，中途半端に書き直すよりはそのまま使った方が速い場合も多い（これはOpenCVに限らず，例えばCの標準ライブラリなどでもいえる）

- 逆に考えると，標準で用意されている関数の実装を見ると，どのように書くと速くなるのかのヒントがいっぱい手に入る

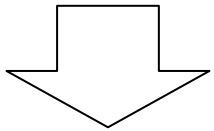
例: さらに計算を減らす

- さて，OpenCV 標準の関数に負けているということは，まだまだ「演算処理」自体が遅いに違いない
- 特に「重い」計算を減らす（乗算・除算・浮動小数点演算）
- アルゴリズム上必要なものはしかたがない
- 画像を走査するときのアドレス計算にまだ改善の余地あり

```
(*(uchar *)((iplimagep)->imageData + (y) * (iplimagep)->widthStep + (x)))
```

ここの乗算を減らしてみる

```
for (j = 0; j < img->height; j++) {
    for (i = 0; i < img->width; i++) {
        (*(uchar *)(img->imageData
                    + j * img->widthStep + i)) = ...
    }
}
```



```
char *ip = img->imageData;
int joffset, offset;

for (j = 0, joffset = 0; j < img->height;
     j++, joffset += img->widthStep) {
    for (i = 0; i < img->width; i += 1) {
        offset = joffset + i;
        *(uchar *)(ip + offset) = ...;
    }
}
```

- サンプル: time_framesub_unroll.cpp
- 予想外なほど効果があった (3倍程度高速化). このくらいはコンパイラがやってくれてもバチは当たらないと思うのだが…?

例: Loop Unrolling

- 繰り返し回数の多いループを，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
- サンプル: time_framesub_unroll.cpp
- わずかにではあるが，効果はあるようだ．このくらいはコンパイラがやって（以下略

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い．今後に注目．