Intelligent Control Systems

Speeding-up Techniques of Image Processing

Shingo Kagami
Graduate School of Information Sciences,
Tohoku University
swk(at)ic.is.tohoku.ac.jp

http://www.ic.is.tohoku.ac.jp/ja/swk/

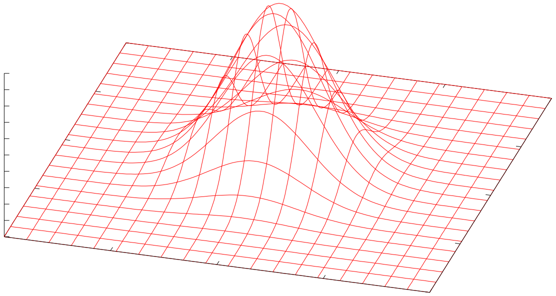
Fast Image Processing

- We have learned basics of image processing and a few standard methods of visual tracking
- In some respects, we have ignored performance issues
    - The same computation may be achieved by different algorithms
    - The same algorithm may become fast or slow depending on the way it is coded

- Bearing in mind real-time applications (e.g. visual servoing), we will learn speeding-up techniques for image processing

Algorithm Choice Example: Gaussian Filter

$$G_{x,y} = \sum_i \sum_j w_{i,j} F_{m+i,n+j}$$



- m× n kernel convolution requires computational time proportional to *mn* for each pixel

- When the kernel is separable as $w_{x,y} = u_x v_y$, the cost becomes proportional to *m* + *n*:

$$G_{x,y} = \sum_i \sum_j u_i v_j F_{m+i,n+j} = \sum_i u_i (\sum_j v_j F_{m+i,n+j})$$

e.g.: $$w_{x,y} = \frac{1}{2\pi\sigma^2} \text{exp}\{-\frac{x^2+y^2}{2\sigma^2}\}$$
$$= \frac{1}{\sqrt{2\pi}\sigma} \text{exp}\{-\frac{x^2}{2\sigma^2}\} \cdot \frac{1}{\sqrt{2\pi}\sigma} \text{exp}\{-\frac{y^2}{2\sigma^2}\}$$

Besides Algorithm Choices

- The most important thing is to choose good algorithms
  - Fast Fourier Transform
  - separable filters
  - nonlinear optimization (vs. full search)
- Even if the same algorithm is used, performance can be significantly affected by implementation

- Let’s see how a simple sample program can be speeded up:

Highlighting frame difference of 640x480 images

- Using OpenCV functions: 1 ~ 2 ms
- Naive Implementation: 2 ~ 3 ms

Highlighting Frame Difference: Algorithm

```
Images input, gray, prev_gray, output;

Repeat {
    // color conversion from BGR to Gray
    for each (i,j) {
        gray(i,j) := BGR2GRAY(input(i,j))
    }
    // take frame difference and highlight
    for each (i,j) {
        output(i,j) :=
          { blue, |gray(i,j) – prev_gray(i,j)| > threshold
          { gray(i,j), otherwise
    }
    // save current frame
    for each (i,j) {
        prev_gray(i,j) := gray(i,j)
    }
}
```

How to Measure Elapsed Time

Using OpenCV functions:

```
double t_begin = (double)cv::getTickCount();
/* the code to be measured */
double t_end = (double)cv::getTickCount();
double delta_in_ms =
  1000.0 * (tick_end - tick_begin) / cv::getTickFrequency();
```

Or, you can use my library *stattimer* (Get stattimer.hpp from http://code.google.com/p/stattimer/ and put it somewhere in your include path):

```
#include "stattimer.hpp"
STimerList st;

st.start("label1");
/* the code to be measured */
st.stop("label1");
```

The results are reported when the program finishes

Outline

- Local Optimization of Coding
- Pixel Access Methods
- Loop Optimization
- Parallel Processing

Common Sense: What are slow?

fast ↔ **slow**

integer operations >>> floating point operations

add, sub, logic >> multiplication >>>>>>>>> division

arithmetic/logic >> jump >>> function call

pipeline hazard stack&register operation overhead

arithmetic/logic >>>>>>>> memory access

local/continuous memory access >>>> global/random access

cache memory principle

mutually-independent instructions >>> dependent instructions

superscalar pipeline principle

Tech. 1: Table Lookup

- If an expensive operations can be done beforehand and the results can be stored in memory, the operations can be replaced by table lookups

```
...  = (r * 306 + g * 601 + b * 117) / 1024;
```

▼

```
...  = (b2gray[b] + g2gray[g] + r2gray[r]) / 1024;
```

Pros: reduces costly operations

Cons: increases memory access

integer operations >>> floating point operations

add, sub, logic >> multiplication >>>>>>>> division

arithmetic/logic >> jump >>> function call

arithmetic/logic >>>>>>> memory access

Tech. 2: Strength Reduction

- The same algorithm may be achieved by weaker (less computationally expensive) operations

```
if (diff > 30 || diff < -30) {
    img.at<cv::Vec3b>(j, i)[0] = 255;
    img.at<cv::Vec3b>(j, i)[1] = 0;
} else {
    img.at<cv::Vec3b>(j, i)[0] = g;
    img.at<cv::Vec3b>(j, i)[1] = g;
}
```

▼

```
int active = ((diff > 30 || diff < -30) && 255);
img.at<cv::Vec3b>(j, i)[0] = g | active;
img.at<cv::Vec3b>(j, i)[1] = g & ~active;
```

add, sub, logic >> multiplication >>>>>>>> division

arithmetic/logic >> jump >>> function call

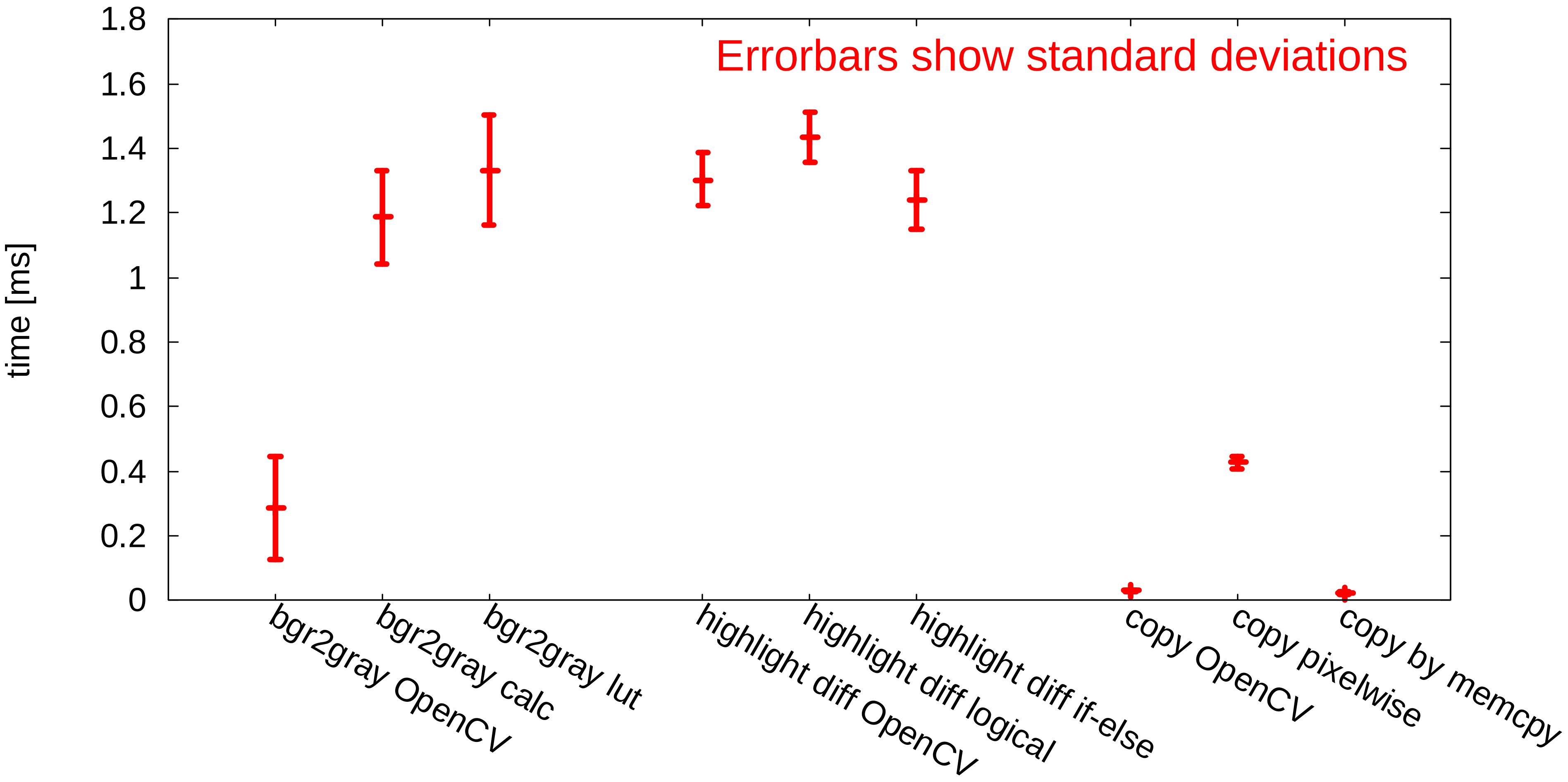
Tech. 3: Bulk Memory Copy

- Instead of copying pixels by iterating through the memory, you can try `memcpy`

- Using this is possible only when the copied data are stored in a continuous area of memory
  - e.g.: To copy a sub rectangle in an image, `memcpy` must be done line by line

- `memset` sometimes will be also useful

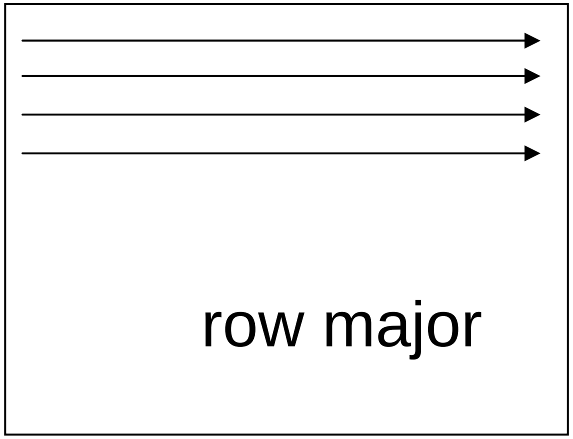
Results (each part)



- Spec: Core i7-4600U 2.1 GHz, 16 GB memory
- Some work better; Some work worse!

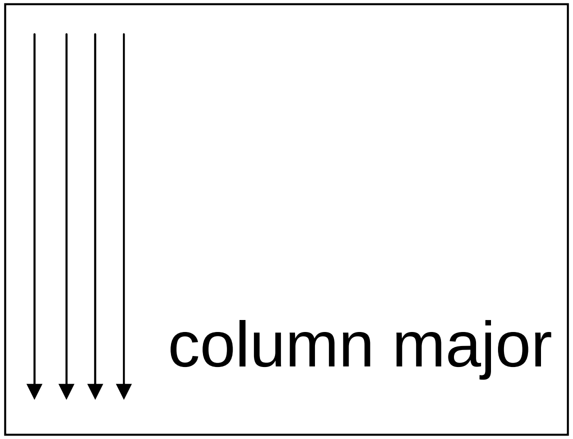
Note: Row or Column Major Access

```
for (j = 0; j < height; j++) {
    for (i = 0; i < width; i++) {
        image.at<uchar>(j, i) = ...

    }
}
```



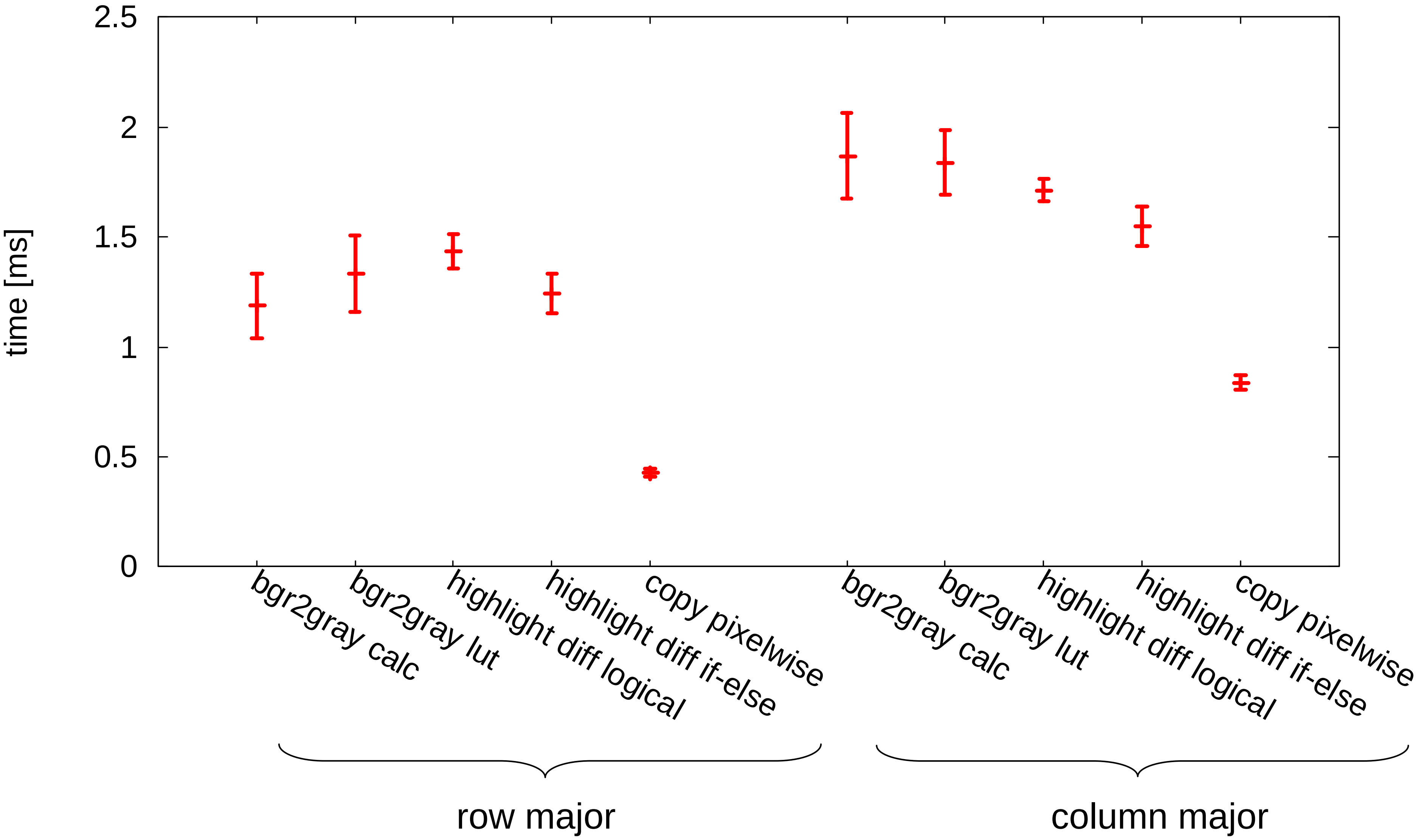
cv::Mat stores data in row-major order

```
for (i = 0; i < width; i++) {
    for (j = 0; j < height; j++) {
        image.at<uchar>(j, i) = ...

    }
}
```



local/continuous memory access >>>> global/random access

Results (each part)



Outline

- Local Optimization of Coding
- Pixel Access Methods
- Loop Optimization
- Parallel Processing

Tech. 4: Pixel Access Methods

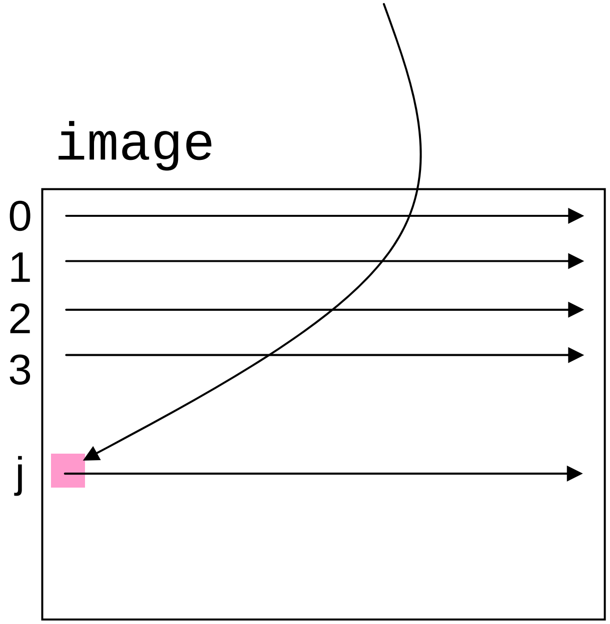
```
for (j = 0; j < height; j++) {
    for (i = 0; i < width; i++) {
        image.at<uchar>(j, i) = ...

    }
}
```

width * j + i

```
for (j = 0; j < height; j++) {
    uchar *ptr = image.ptr<uchar>(j);
    for (i = 0; i < width; i++) {
        ptr[i] = ...
    }
}
```

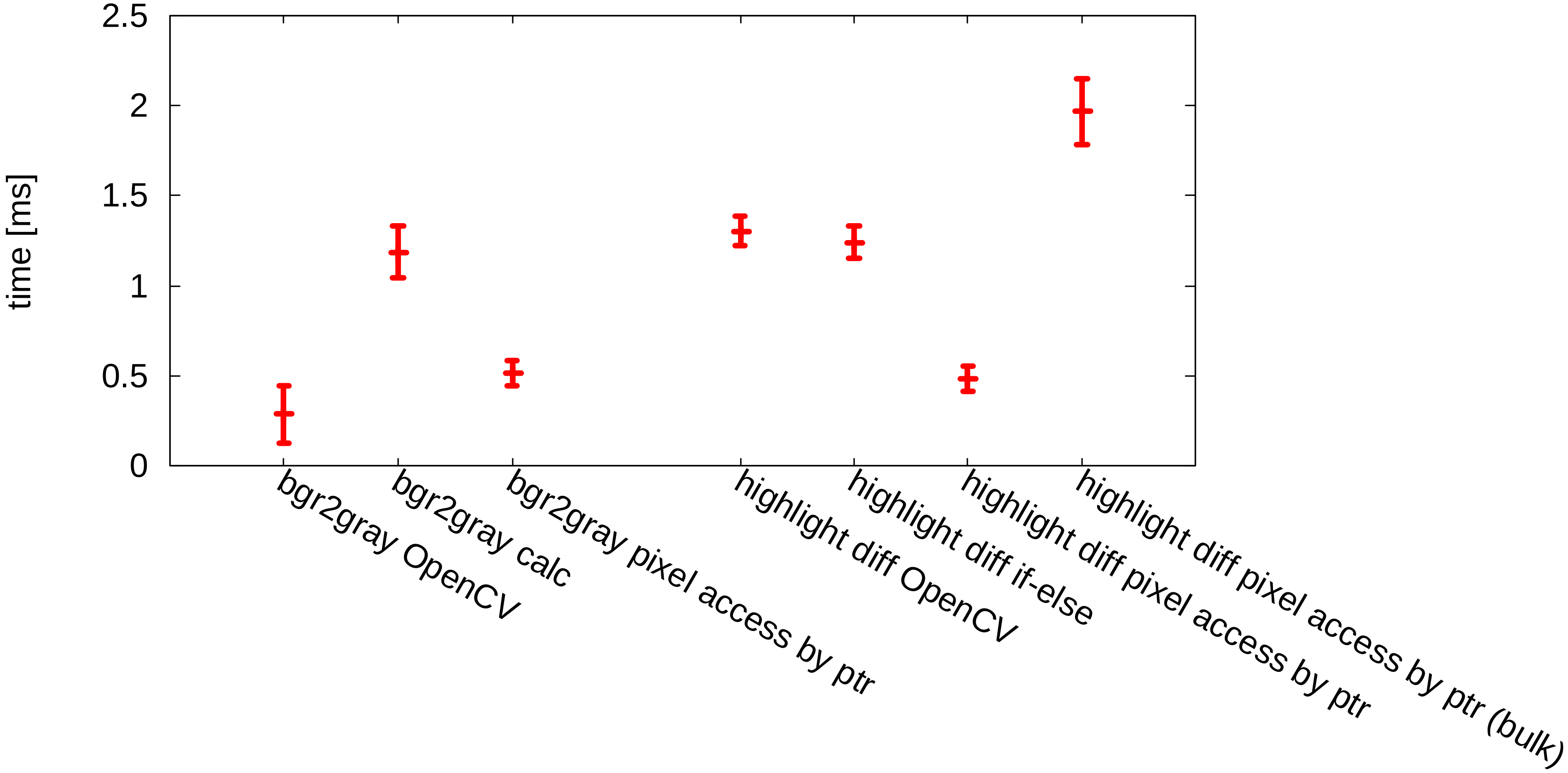
ptr + i

`image.ptr<..>(j)`



add, sub, logic >> multiplication >>>>>>>>> division

Results (each part)



In my environment (Visual Studio 2012), treating a color pixel (cv::Vec3b)
in bulk like `outp[i] = cv::Vec3b(255, 0, 0);`
is ridiculously slow

Outline

- Local Optimization of Coding
- Pixel Access Methods
- Loop Optimization
- Parallel Processing

Tech. 5: Loop Fusion

```
for (j = 0; j < height; j++) {
    for (i = 0; i < width; i++) {
        f(...);
    }
}
for (j = 0; j < height; j++) {
    for (i = 0; i < width; i++) {
        g(...);
    }
}
```

```
for (j = 0; j < height; j++) {
    for (i = 0; i < width; i++) {
        f(...);
        g(...);
    }
}
```

- smaller loop overheads
- improved memory locality
- more independent instructions within a loop

local/continuous memory access >>>> global/random access

mutually-independent instructions >>> dependent instructions

Tech. 6: Loop Unrolling

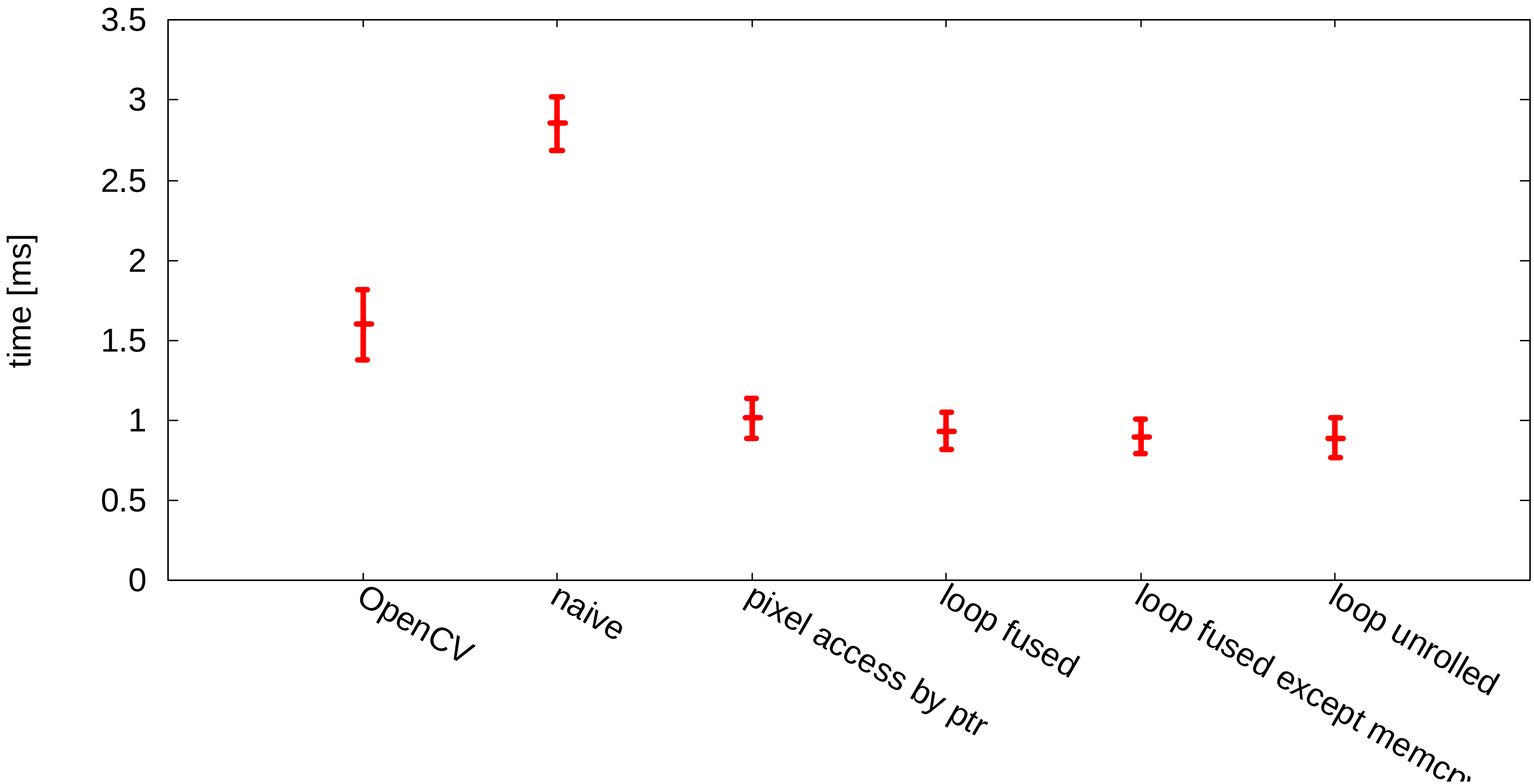
```
for (i = 0; i < N; i++) {
    f(i, ...) = ...
}
```

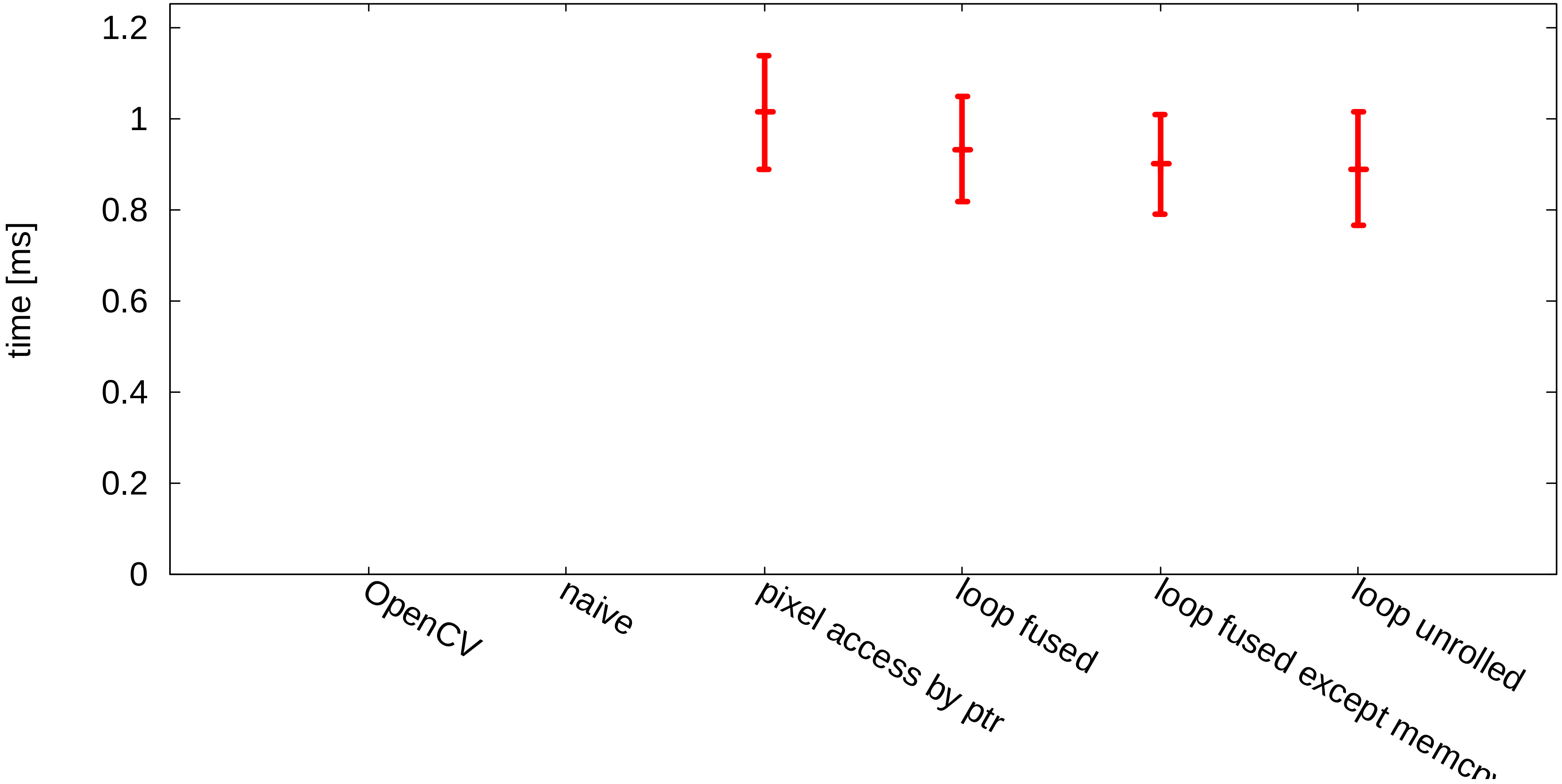
```
for (i = 0; i < N; i += 4) {
    f(i, ...) = ...
    f(i+1, ...) = ...
    f(i+2, ...) = ...
    f(i+3, ...) = ...
}
```

- smaller loop overheads
- more independent instructions within a loop

mutually-independent instructions >>> dependent instructions

Results (total)





time [ms]
1.2
1
0.8
0.6
0.4
0.2
0
OpenCV
naive
pixel access by ptr
loop fused
loop fused except memcp
loop unrolled

Outline

- Local Optimization of Coding
- Pixel Access Methods
- Loop Optimization
- Parallel Processing

Tech. 7: Multi-Threading

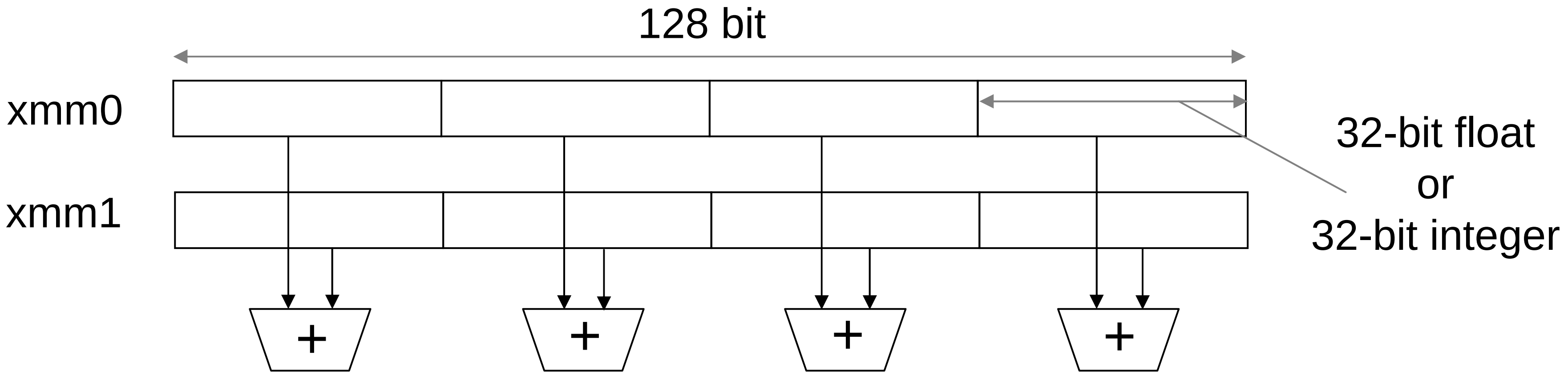
- Image processing in general has high data parallelism, and thus parallel processing is effective
- One of the easiest way is to parallelize `for` loops into multiple threads using OpenMP
  - Threads will be executed in multiple cores
  - Visual C++ supports OpenMP by default
    - Config. properties – C/C++ – Language – OpenMP

```
#pragma omp parallel for num_threads(4)
for (j = 0; j < height; j++) {
    for (i = 0; i < width; i++) {
        image.at<uchar>(j, i) = ...
    }
}
```

Tech. 8: SIMD Extensions

- SIMD: Single Instruction stream, Multiple Data stream
  cf. MIMD
- Many recent processors have extended instruction set to perform SIMD operations
  - MMX, SSE, AVX (intel)

- In SSE, eight 128-bit registers (xmm0, ... xmm7) are used
  - sixteen 8-bit data, eight 16-bit data, four 32-bit data, or two 64-bit data are processed at a time



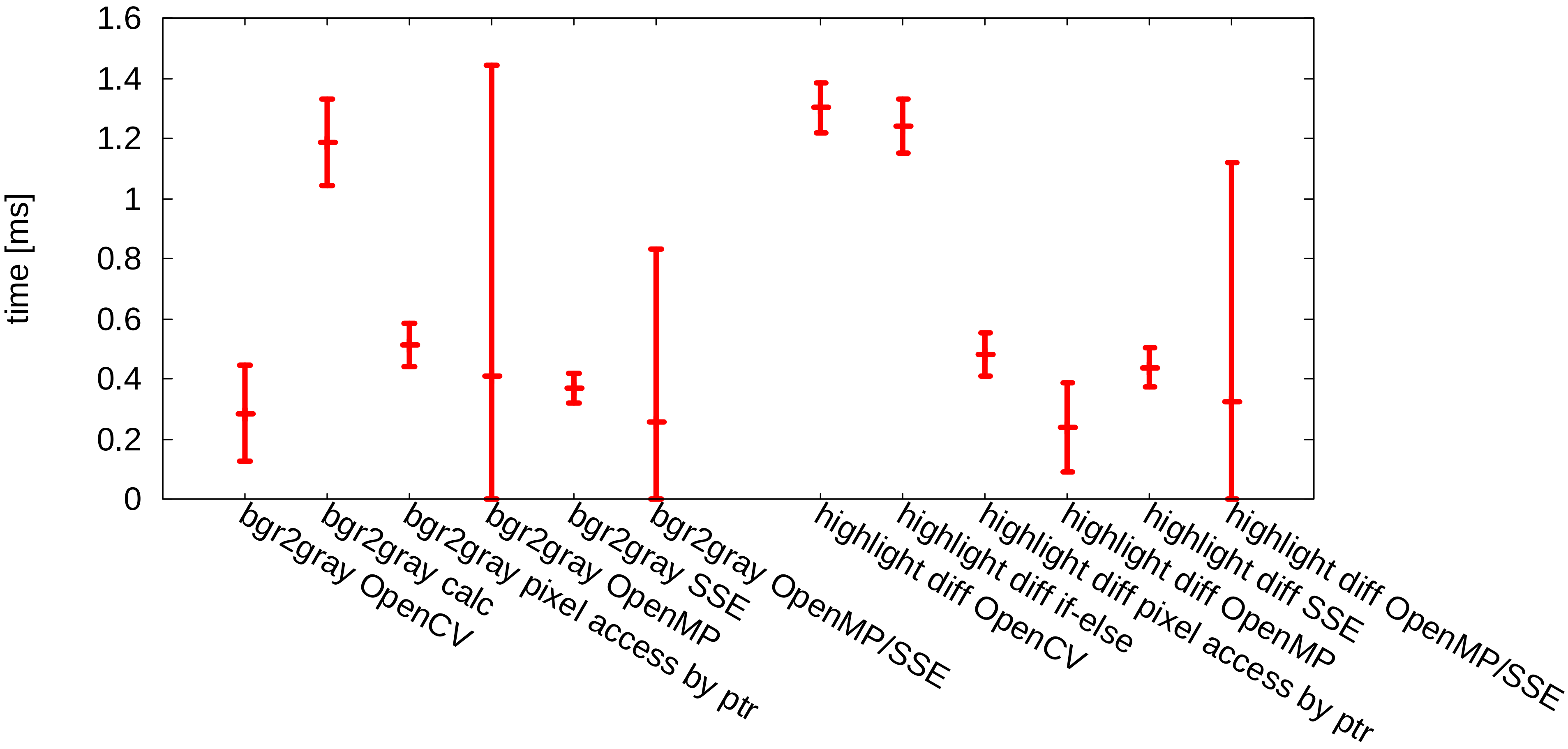
Compiler intrinsics: easiest way to explicitly use SSE

- common for Visual C++ and GCC

```
float sum = 0.0f;
for (i = 0; i < N; i++) {
    sum += w[i] * x[i];
}
```

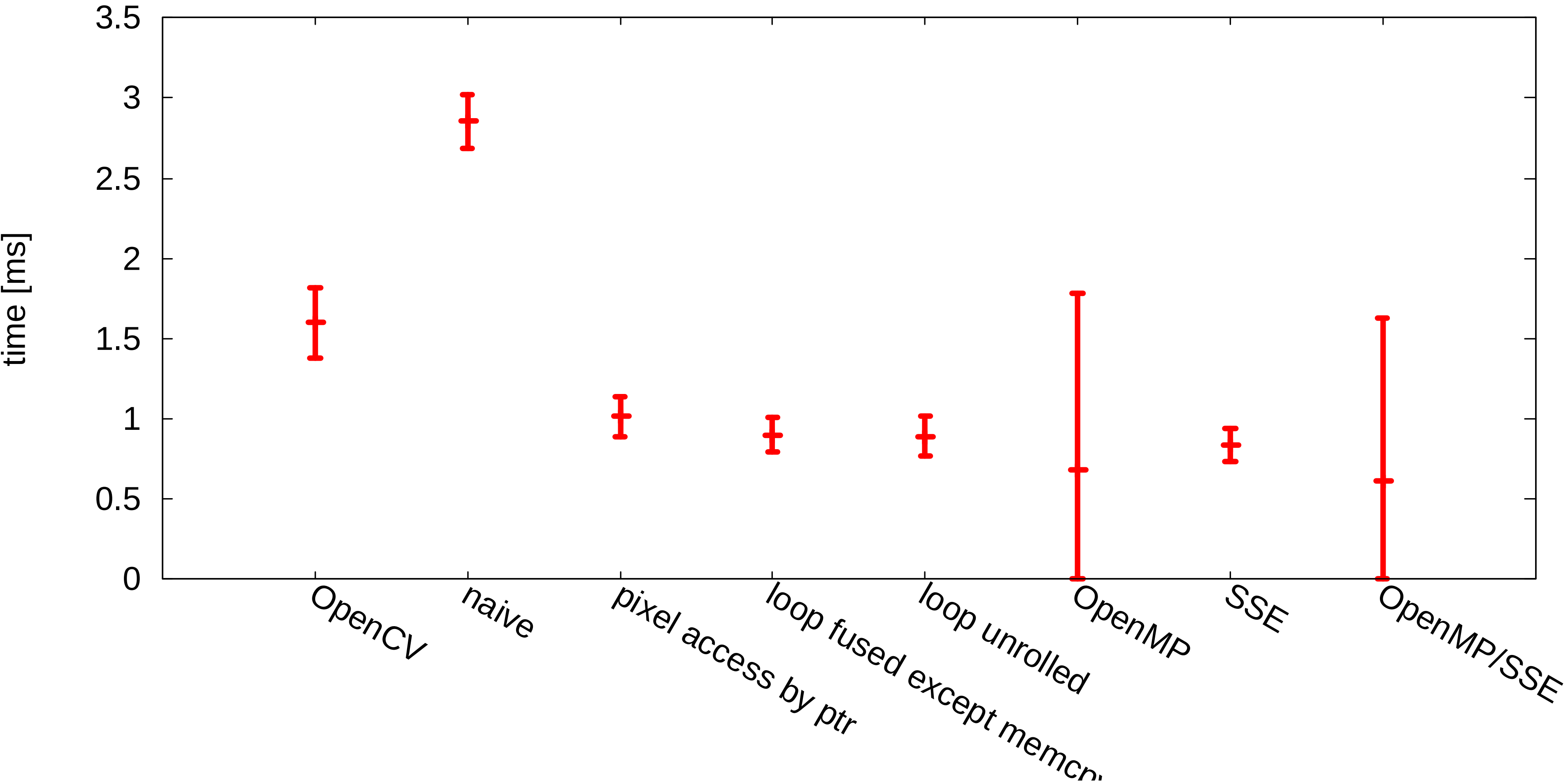
```
__m128 sum = _mm_setzero_ps();
for (i = 0; i < N; i += 4) {
    __m128 ws = _mm_loadu_ps(&w[i]);
    __m128 xs = _mm_loadu_ps(&x[i]);
    sum = _mm_add_ps(sum, _mm_mul_ps(ws, xs));
}
...
```

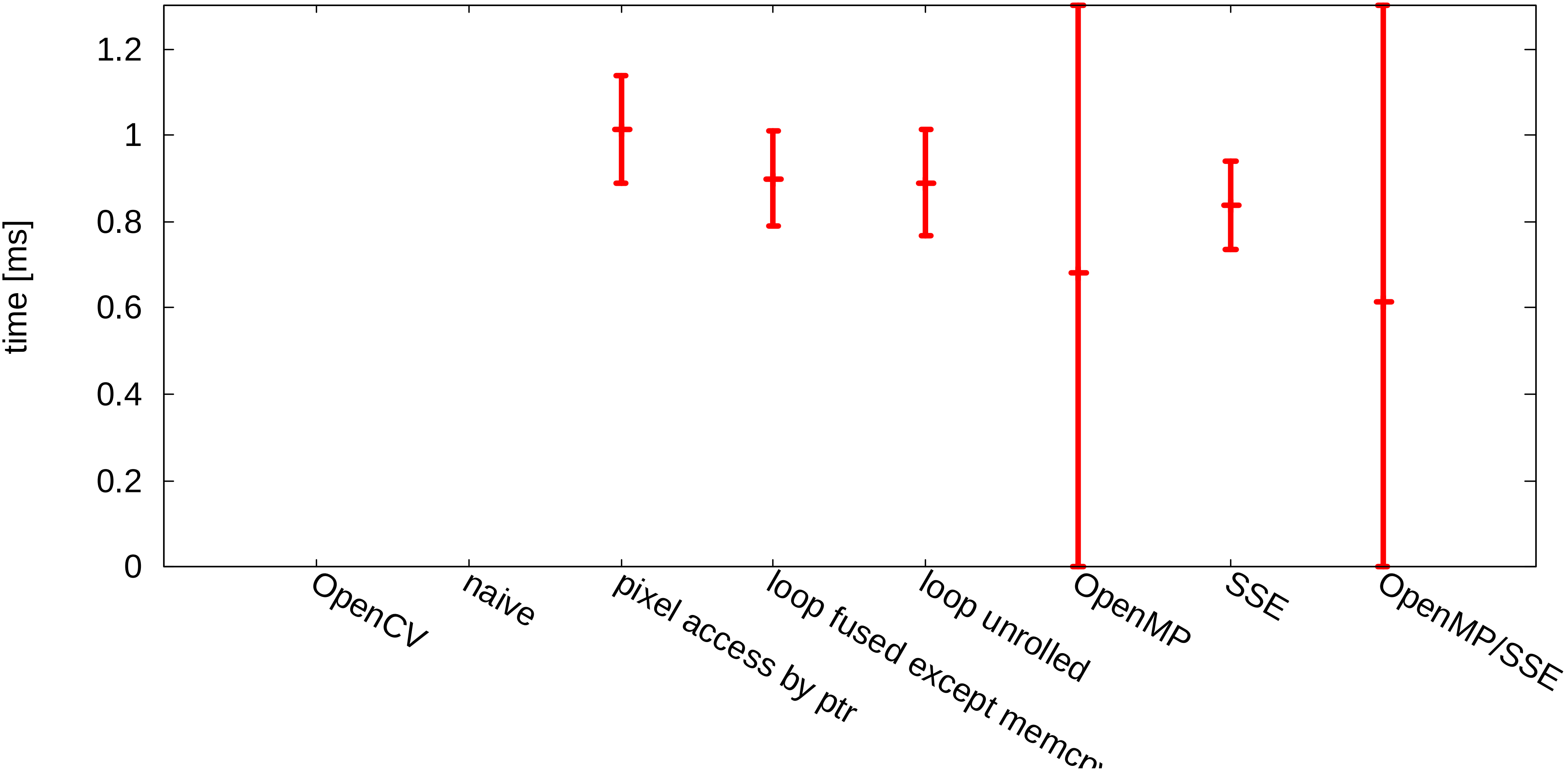
Results (each part)



Note the large deviation when OpenMP is enabled

Results (total)




time [ms]
0
0.2
0.4
0.6
0.8
1
1.2
OpenCV
naive
pixel access by ptr
loop fused except memcp
loop unrolled
OpenMP
SSE
OpenMP/SSE

Summary

- Pixel Scan Order
  - Pixels should be accessed in the order in which they are stored (row major in OpenCV)
- Pixel Access Methods
  - `at()` is slow! Using `ptr()` instead significantly improves the performance
- Other Optimizations
  - strength reduction, table lookup, loop fusion, loop unrolling
- Parallel Processing
  - OpenMP, SIMD extension, (GPU was not mentioned today)
- Some work fine; Some do not (Some may work even worse)
  - Trial & error are needed
  - Trade-off between performance and maintainability
  - Too early optimization should be avoided

References

- D. Bulka and D. Mayhew: Efficient C++: Performance Programming Techniques, Addison-Wesley, 1999.
- http://code.google.com/p/stattimer/ (as of 2014/7/22)

(in Japanese)
- 片山: Cプログラム高速化研究班, USP研究所, 2012.

Sample codes are in sample20140722.zip available at
http://www.ic.is.tohoku.ac.jp/~swk/lecture/