

Basic code optimization

Fabien Spindler

Lagadic team

INRIA Rennes-Bretagne Atlantique - IRISA

<http://www.irisa.fr/lagadic>

Seminar, June 23-24, 2009



lagadic



 INRIA

The INRIA logo, featuring a stylized white 'R' inside a square, followed by the letters "INRIA" in a white, sans-serif font.

Outline

1. Why
2. What
3. How



1. Why

Why should we optimized a code?

- Reduce the binary size
- Consume less memory
- Increase the time performance

Focus on time optimization for real time applications

Prerequisites

- The code is tested and validated
- Objectives are defined
- Profiling tools are available



2. What

To identify the critical functions to optimize use a profiling tool

Profiling allows to identify

- Time consummation for a function
- Number of calls per function

Profiling tools

- GNU gprof (add `-g -pg` compiler options)

```
> g++ -g -pg -o gaussian-filter gaussian-filter.cpp  
> ./gaussian-filter  
> gprof -b gaussian-filter > profile.out
```

- Valgrind-Cachegrind (works in release mode)

```
> g++ -O3 -o gaussian-filter gaussian-filter.cpp  
> valgrind --tool=cachegrind ./gaussian-filter  
> kcachegrind
```



3. How

Let us consider 3 examples implemented in ViSP (vpImageFilter):

- Gaussian 5x5 filter
- Derivative 1x7 filter along x axis
- Derivative 7x1 filter along y axis



Case 1: ViSP Gaussian 5 by 5 filter

Implemented in:

```
static double  
vpImageFilter::gaussianFilter(vpImage<uchar> &I, int i, int j)
```

$$G[i][j] = \sum_{m=-2}^2 \sum_{n=-2}^2 F[m][n] I[i-m][j-n] \quad F = \frac{1}{159} \begin{bmatrix} 2 & 4 & 5 & 4 & 2 \\ 4 & 9 & 12 & 9 & 4 \\ 5 & 12 & 15 & 12 & 5 \\ 4 & 9 & 12 & 9 & 4 \\ 2 & 4 & 5 & 4 & 2 \end{bmatrix}$$

Usage:

```
for (int i=0; i < I.getHeight(); i++)  
  for (int j=0; j < I.getWidth(); j++)  
    G[i][j] = vpImageFilter::gaussianFilter(I, i, j);
```



Case 2: ViSP derivative filter along x

Implemented in:

```
static double  
vpImageFilter::derivativeFilterX(vpImage<uchar> &I, int i, int j)
```

$$D_x[i][j] = \sum_{n=-2}^2 F_x[n] I[i][j - n]$$

$$F_x = \frac{1}{8418} \begin{bmatrix} -112 & -913 & -2047 & 0 & 2047 & 913 & 112 \end{bmatrix}$$

Usage:

```
for (int i=0; i < I.getHeight(); i++)  
  for (int j=0; j < I.getWidth(); j++)  
    G[i][j] = vpImageFilter::derivativeFilterX(I, i, j);
```



Case 3: ViSP derivative filter along y

Implemented in:

```
static double  
vpImageFilter::derivativeFilterY(vpImage<uchar> &I, int i, int j)
```

$$D_y[i][j] = \sum_{m=-2}^2 F_y[m] I[i-m][j]$$
$$F_y = \frac{1}{8418} \begin{bmatrix} -112 \\ -913 \\ -2047 \\ 0 \\ 2047 \\ 913 \\ 112 \end{bmatrix}$$

Usage:

```
for (int i=0; i < I.getHeight(); i++)  
  for (int j=0; j < I.getWidth(); j++)  
    G[i][j] = vpImageFilter::derivativeFilterY(I, i, j);
```



Optimization done by the compiler

Testbed:

- viper: two Intel Core 2 E8400 @3Ghz, Fedora 10
- Image size: 640 x 480
- Compilers: GNU g++ 4.3.0 and Intel icc 10.1

Focus on default CMake configuration

- On Unix, it is important to set the build type during CMake configuration

<code>CMAKE_BUILD_TYPE = None or Debug</code>	<code>g++</code>	36105 us
<code>CMAKE_BUILD_TYPE = Release</code>	<code>g++ -O3</code>	8378 us

- On unix the default compiler is GNU g++
- To use Intel icc, set CXX environment variable by:
 - `setenv CXX icpc` Or `export CXX=icpc`
- To modify compiler options, in CMake GUI modify `CMAKE_CXX_FLAGS_RELEASE`



Optimization done by the compiler

Comparison between GNU g++ and Intel icc

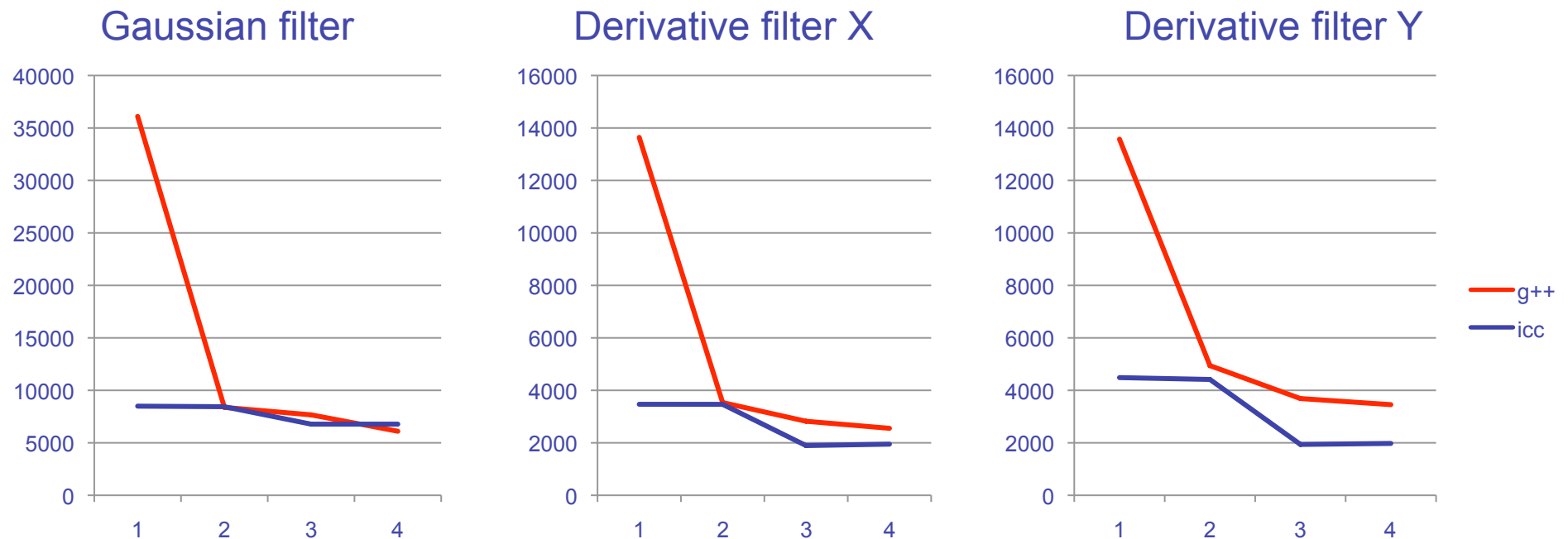
	Gaussian filter	Time
(1)	g++	36105 us
(2)	g++ -O3	8378 us
(3)	g++ -O3 -ffast-math	7666 us
(4)	g++ -O3 -ffast-math -funroll-loops -fomit-frame-pointer -mfpmath=sse -march=core2	6104 us
(1)	icpc	8494 us
(2)	icpc -O3	8442 us
(3)	icpc -O3 -fast	6781 us
(4)	icpc -O3 -fast -funroll-loops -fomit-frame-pointer	6782 us

In this example GNU g++ is more efficient than icc



Optimization done by the compiler

Comparison between GNU g++ and Intel icc



GNU g++ and Intel icc fast options help a lot

It seems not possible to conclude that Intel icc is generally better

Is it possible to optimize more by hand ? Profiling may help now



Example of GNU gprof output

```
% cumulative self self total
time seconds seconds calls ms/call ms/call name
68.75 0.06 0.06 7402500 0.00 0.00 vpImage<unsigned char>::operator[](unsigned int)
18.75 0.07 0.01
main
12.50 0.08 0.01 296100 0.00 0.00 vpImageFilter::gaussianFilter(vpImage<unsigned char>&, int, int)
0.00 0.08 0.00 603771 0.00 0.00 vpImage<unsigned char>::getWidth() const
0.00 0.08 0.00 307672 0.00 0.00 vpImage<unsigned char>::getHeight() const
0.00 0.08 0.00 296100 0.00 0.00 vpImage<double>::operator[](unsigned int)
0.00 0.08 0.00 2 0.00 0.00 measureTimeMicro()
0.00 0.08 0.00 1 0.00 65.00 gaussian_original(vpImage<unsigned char>&, vpImage<double>&)
0.00 0.08 0.00 1 0.00 0.00 vpImage<double>::init(unsigned int, unsigned int)
0.00 0.08 0.00 1 0.00 0.00 vpImage<double>::init(unsigned int, unsigned int, double)
0.00 0.08 0.00 1 0.00 0.00 vpImage<double>::destroy()
0.00 0.08 0.00 1 0.00 0.00 vpImage<double>::vpImage(unsigned int, unsigned int)
0.00 0.08 0.00 1 0.00 0.00 vpImage<double>::~~vpImage()
0.00 0.08 0.00 1 0.00 0.00 vpImage<unsigned char>::init(unsigned int, unsigned int)
0.00 0.08 0.00 1 0.00 0.00 vpImage<unsigned char>::init(unsigned int, unsigned int, unsigned char)
0.00 0.08 0.00 1 0.00 0.00 vpImage<unsigned char>::destroy()
0.00 0.08 0.00 1 0.00 0.00 vpImage<unsigned char>::vpImage(unsigned int, unsigned int)
0.00 0.08 0.00 1 0.00 0.00 vpImage<unsigned char>::~~vpImage()
```



Code parallelization with OpenMP (opt 1)

OpenMP is an Application Program Interface (API) that may be used to explicitly direct multi-threaded, shared memory parallelism

How to use

- Introduce pragma in the code
- Link with OpenMP library

Example

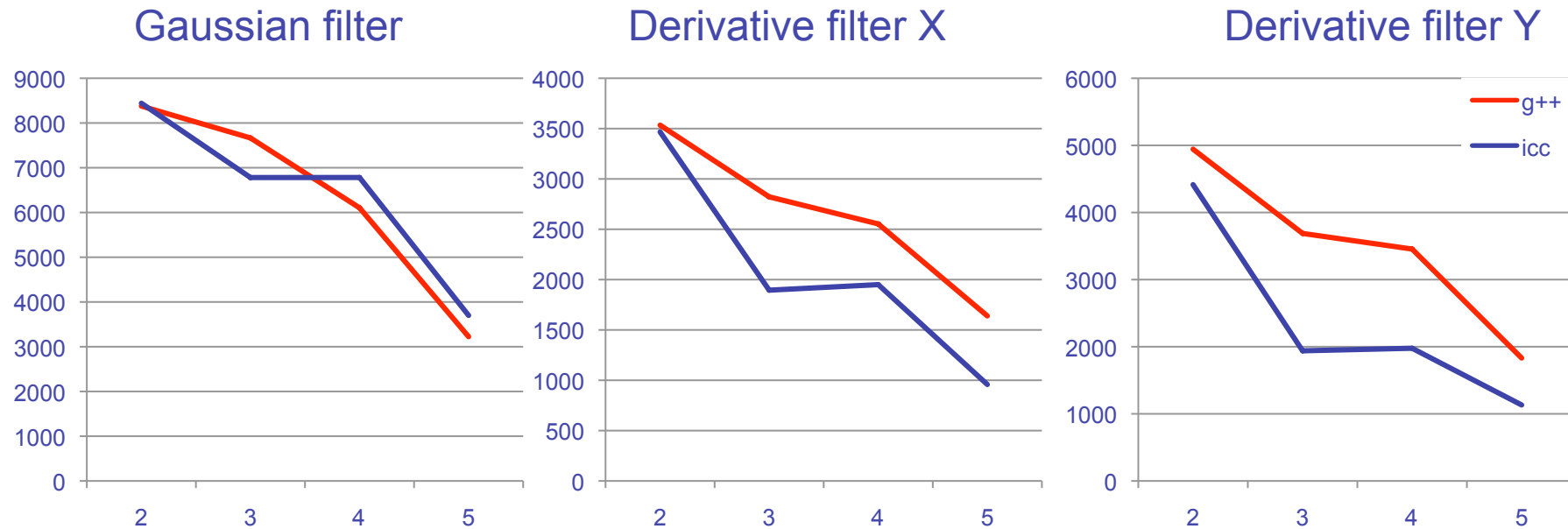
```
#pragma omp parallel for
for (int i=0; i < I.getHeight(); i++)
  for (int j=0; j < I.getWidth(); j++)
    G[i][j] = vpImageFilter::gaussianFilter(I, i, j);
```

(4)	g++ -O3 -ffast-math ...	6104 us
(5)	g++ -O3 -ffast-math ... -fopenmp -lgomp	3699 us
(4)	icpc -O3 -fast ...	6782 us
(5)	icpc -O3 -fast ... -openmp	3224 us



Optimization done by the compiler

Comparison between GNU g++ and Intel icc



Code threading can be done at low cost with OpenMP

Is it possible to optimize more by doing basic things?



lagadic

INRIA

14

Hand made optimization

Optimization (6): inline small atomic functions

- An inline function is substituted by its body
- Change the function declaration by adding the C++ key world “inline”

```
static inline double vpImageFilter::gaussianFilter(vpImage<uchar> ...)
```

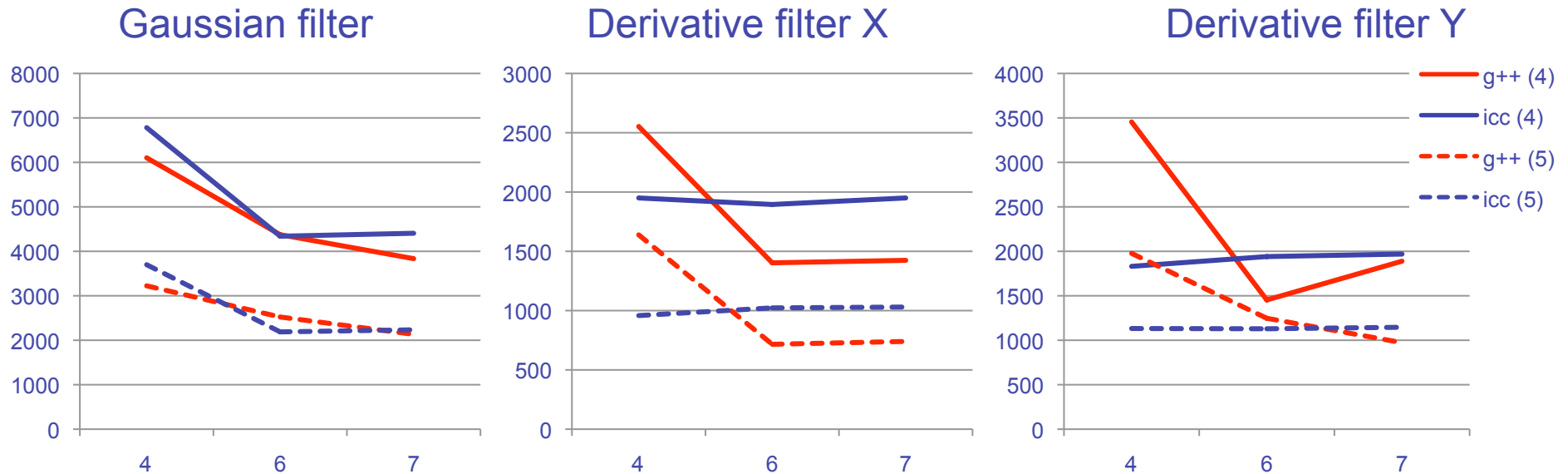
Optimization (7): often called functions from within a loop

- Change the loop to reduce the overhead of calling the function repeatedly
- should be very similar to inline

```
vpImageFilter::derivativeFilterXNew(I, G) { ...  
    int height = I.getHeight();  
    int width  = I.getWidth() - border;  
    #pragma omp parallel for  
    for (int i=0; i < height; i++)  
        for (int j=border; j < width; j++)  
            G[i][j] = ( 2047.0 *(I[i][j+1] - I[i][j-1])  
                + 913.0 *(I[i][j+2] - I[i][j-2])  
                + 112.0 *(I[i][j+3] - I[i][j-3]) ) / 8418.0;  
}
```



Hand made optimization



In these examples it is possible to do better than the compiler by:

- Inlining small functions (very simple to do)
- Modifying the loop to avoid often called functions from within a loop



Conclusion

Time for viper and (quipu)	Gaussian	Derivative X	Derivative Y
g++ -O3 (standard release)	8378 (4334) us	3534 (2414) us	4943 (2661) us
g++ with all optimizations	2232 (382) us	740 (149) us	976 (144) us
icc -O3 (standard release)	8442 us	3467 us	4415 us
icc with all optimizations	2188 us	1023 us	1129 us

Do not optimize too early

Depending on your real time constraints

- Choose the appropriate compiler and compiler options
- Introduce OpenMP pragma

Identify critical code

- Hand made optimization may help
- How long?
- Compiler will be more and more efficient

