SMIL : Simple Morphological Image Library

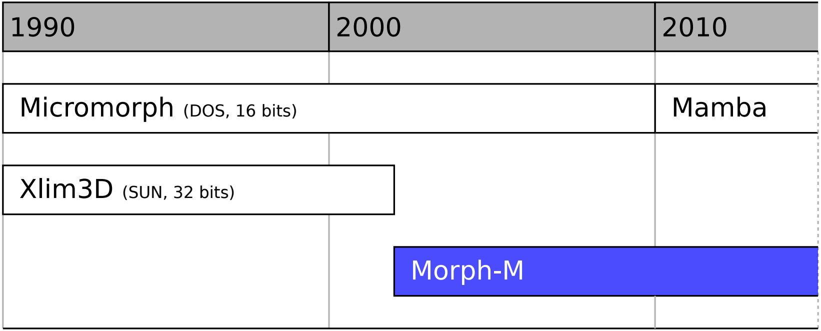
Matthieu Faessel, Michel Bilodeau

Centre de Morphologie Mathématique, Mathématiques et Systèmes,
MINES ParisTech

Séminaire LRDE, 27/03/2013

Table of contents

CMM image libraries history



Morph-M: a research library

Advantages: suitable for exploration

- Generic (gray, color, multispectral images, graphs, . . .)
- N-Dimensions
- Meta-programming
- A lot of contributions (common workspace for students, researchers, . . .)

Drawbacks

- Developement time
- Clarity of the code (for users and contributors): a high level of abstraction
- **Performances**: not in the initial specifications, very difficult to integrate afterwards
- Proprietary Licence: not very flexible for industrial projetcs

Morph-M: a research library

Advantages: suitable for exploration

- Generic (gray, color, multispectral images, graphs, . . .)
- N-Dimensions
- Meta-programming
- A lot of contributions (common workspace for students, researchers, . . .)

Drawbacks

- Developement time
- Clarity of the code (for users and contributors): a high level of abstraction
- **Performances**: not in the initial specifications, very difficult to integrate aferwards
- Proprietary Licence: not very flexible for industrial projetcs

New constraints

Industrial projects with performance constraints increasingly strong

- Real-time applications (ex: industrial control, video surveillance)
- Near real-time processing of large 2D images (ex: $40.000 \times 1.200/sec$)
- 3D images (ex: $1.000^3 \rightarrow 1\ GPix$)
- Embedded systems

New needs

- Very fast algoritms / Support for large data
- Relatively light
- Easy prototyping
- Ease of integration

New constraints

Industrial projects with performance constraints increasingly strong

- Real-time applications (ex: industrial control, video surveillance)
- Near real-time processing of large 2D images (ex: $40.000 \times 1.200/sec$)
- 3D images (ex: $1.000^3 \rightarrow 1\ GPix$)
- Embedded systems

New needs

- Very fast algoritms / Support for large data
- Relatively light
- Easy prototyping
- Ease of integration

Fast Morph-M alternatives

Mamba

Pros:

✓ Fast (intrinsic SIMD functions)

✓ Easy to use (python, embedded viewer, . . .)

✓ Implementation of new python functions is simple

Cons:

✗ Most functions are written in python (with dependency with other python libraries) ⇒Problems for integration

✗ C based Core: no factorization (much redundant code)⇒Rigidity, not suitable for general improvements or newer developpements

✗ SIMD intrinsic functions: a lot of code lines

Fast Morph-M alternatives

Fulguro

Pros:

✓ Fast (intrinsic SIMD functions)

✓ C-style factorization (macros)

Cons:

✗ C-style factorization (macros)

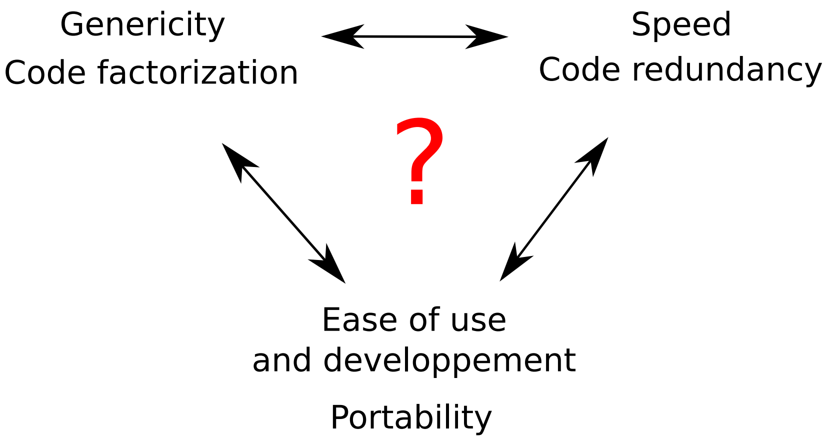
✗ SIMD intrinsic functions: a lot of code lines

Fast-MorphM

MorphM SIMD Addon

Very few functions

A solution ?


Genericity
Code factorization
Speed
Code redundancy
?
Ease of use
and developpement
Portability

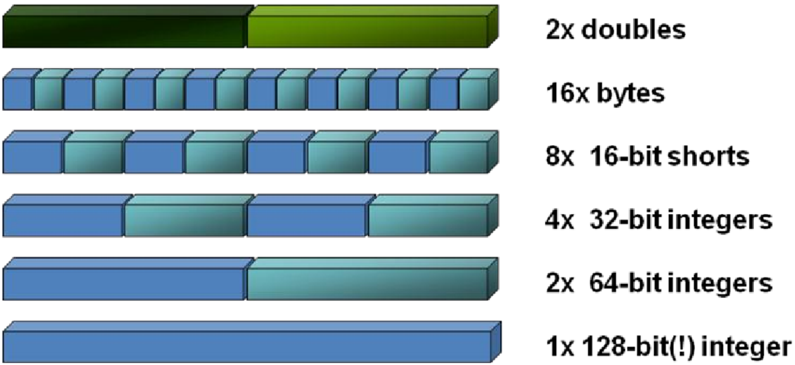
A new done: Auto-Vectorization

GCC 4.2 new feature: auto-vectorization... reconciliates C++ and SIMD

⇒Start a new library from scratch, using auto-vectorization

Vectorization

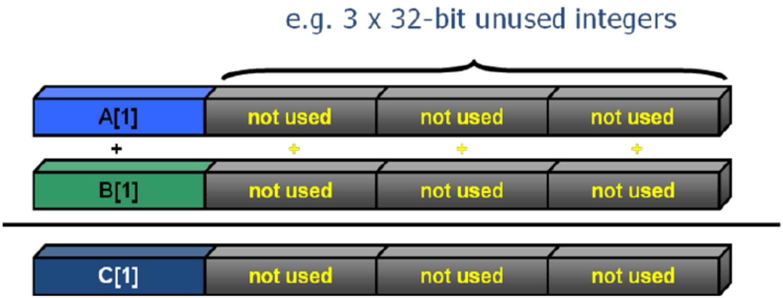
SIMD Registers



Vectorization

```
for(i=0;i<=MAX;i++)
    c[i]=a[i]+b[i];
```

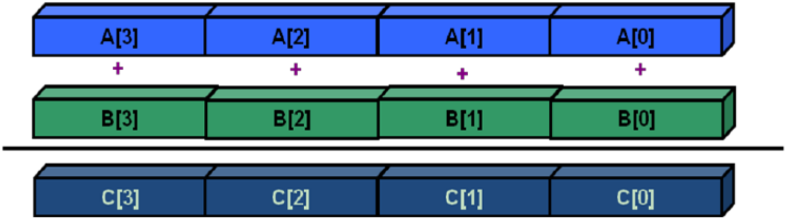
Not vectorized



Vectorization

```
for(i=0;i<=MAX;i++)
    c[i]=a[i]+b[i];
```

Vectorized



How to vectorize ?

- Until auto-vectorization: Intrinsic implementation
  - Requires aligned and contiguous data
  - 1 implementation/data type
  - 1 implementation/SIMD instruction type (SSE, SSE2, . . .)
- With auto-vectorization
  - Requires auto-vectorization capable compiler: GCC ($\geq$4.2), ICC, XLC, CLang, MSVC ($\geq$2012), . . .
  - Requires ~~aligned and~~ contiguous data
  - Requires some code conditions: write vectorizer-friendly code:
    - Countable loop
    - Avoid aliasing problems (single entry and single exit)
    - Straight-line code
    - The innermost loop of a nest
    - No function calls

Auto-vectorization Example

```
\    #define N 128
1.   int a[N], b[N];
2.   void foo (void)
3.   {
4.   int i;
5.
6.   for (i = 0; i < N; i++)
7.     a[i] = i;
8.
9. for (i = 0; i < N; i+=5)
10.        b[i] = i;
11. }
```

```
ex.c:7: note: LOOP VECTORIZED.
ex.c:3: note: vectorized 1 loops in function.
ex.c:10: note: not vectorized: complicated access pattern.
```

Generated Assembler Code

SSE2 vecteurs et scalaires

```
void supLine(const PIXEL *pIn1,
             const PIXEL *pIn2,
             PIXEL *pOut, int count){
   {
   for (int i=0; i< count; i++){

       pOut[i] =
          MAX(pIn1[i], pIn2[i]);
   }
}
```

```
L4:
movdqu (%rsi,%rax), %xmm0
addl $1, %r8d
movdqu (%rdi,%rax), %xmm1
pmaxub %xmm1, %xmm0
movdqu %xmm0, (%rdx,%rax)
addq $16, %rax
cmpl %r8d, %r9d
    ja L4
```

```
L6:
movzbl (%rdi,%rax), %edx
movzbl (%r10,%rax), %esi
cmpb %dl, %sil
cmovae %esi, %edx
movb %dl, (%r9,%rax)
addq $1, %rax
leal (%r8,%rax), %edx
cmpl %edx, %ecx
    jg L6
```

Generated Assembler Code

SSE2

```
void supLine(const PIXEL *pIn1,
             const PIXEL *pIn2,
             PIXEL *pOut, int count){
   {
   for (int i=0; i< count; i++){

       pOut[i] =
          MAX(pIn1[i], pIn2[i]);
   }
}
```

```
L4:
movdqu (%rsi,%rax), %xmm0
addl $1, %r8d
movdqu (%rdi,%rax), %xmm1
pmaxub %xmm1, %xmm0
movdqu %xmm0, (%rdx,%rax)
addq $16, %rax
cmpl %r8d, %r9d
 ja L4
```

Generated Assembler Code

vmax

```
void supLine(const PIXEL *pIn1,
             const PIXEL *pIn2,
             PIXEL *pOut, int count){
   {
   for (int i=0; i< count; i++){

       pOut[i] =
          MAX(pIn1[i], pIn2[i]);
   }
}
```

```
L4:
vmovdqu (%rsi,%rax), %xmm1
addl $1, %r8d
vmovdqu (%rdi,%rax), %xmm0
vpmaxub %xmm0, %xmm1, %xmm0
vmovdqu %xmm0, (%rdx,%rax)
addq $16, %rax
cmpl %r8d, %r9d
    ja L4
```

Generated Assembler Code

vmax2

```
void supLine(const PIXEL *pIn1,
             const PIXEL *pIn2,
             PIXEL *pOut, int count){
   {
   for (int i=0; i< count; i++){

       pOut[i] =
          MAX(pIn1[i], pIn2[i]);
   }
}
```

```
L16:
vmovdqu (%rsi,%rax), %xmm1
addl $1, %r8d
vmovdqu (%rdi,%rax), %xmm0
vinserti128 $0x1, 16(%rsi,%rax), %ymm1, %ymm1
vinserti128 $0x1, 16(%rdi,%rax), %ymm0, %ymm0
vpmaxub %ymm0, %ymm1, %ymm0
vmovdqu %xmm0, (%rdx,%rax)
vextracti128 $0x1, %ymm0, 16(%rdx,%rax)
addq $32, %rax
cmpl %r8d, %r9d
ja L16
```

Generated Assembler Code

boucle while

```
void supLine(const PIXEL *pIn1,
             const PIXEL *pIn2,
             PIXEL *pOut,
             int count)
 {

    while(count--) {

        *pOut++  =
            MAX(*pIn1, *pIn2);
        pIn1++;pIn2++;
    }
}
```

```
L16:
vmovdqu (%rsi,%rax), %xmm1
addl $1, %r8d
vmovdqu (%rdi,%rax), %xmm0
vpcmpgtq %xmm0, %xmm1, %xmm2
vpblendvb %xmm2, %xmm1, %xmm0, %xmm0
vmovdqu %xmm0, (%rdx,%rax)
addq $16, %rax
cmpl %r8d, %r9d
ja L16
```

Generated Assembler Code

boucle while avec fonction compteur

```
int countDecr(int count){
    return count-1;
}
void supLine(const PIXEL *pIn1,
             const PIXEL *pIn2,
             PIXEL *pOut,
             int count)
 {

    while(count) {

        *pOut++  =
            MAX(*pIn1, *pIn2);
        pIn1++;pIn2++;
        count = countDecr(count);
    }
}
```

```
L17:
movq (%r12,%rbx), %rcx
movl %eax, %edi
movq 0(%r13,%rbx), %r8
cmpq %rcx, %r8
cmovge %r8, %rcx
movq %rcx, 0(%rbp,%rbx)
addq $8, %rbx
call __Z9countDecri
testl %eax, %eax
jne L17
```

1. Introduction
2. SMIL: General presentation
3. Optimizations
4. Benchmarks/Perspectives

SMIL: General

Base Objectives

- Speed
- Flexible licence
- Ease of integration:
  - Light
  - No major dependancy
  - Multi-platforms/compilers

Additionnal Objectives

- Easy to use
- Easy to develop/extend
- A lot of factorization:
  - Less code to write
  - Facilitates major evolutions
  - Concentrate optimizations parts

SMIL: Ingredients

- C++
- Templates, but no real genericity (non-standard types require specializations)
- Auto-vectorization
- OpenMP
- CMake
- Swig
- (doxygen, git)
- BSD Licence

Templates and Vectorization

Intrinsic Specializations

Sup_....cpp

#ifdef __SSE__

Sup_UINT16.cpp

#ifdef __SSE__

Sup_UINT8.cpp

```
#ifdef __SSE__
  template<>
  inline void t_LineArithSup1D<UINT8>(...) {
   int i;
   __m64 r0,r1;

   for(i=0 ; i<=size ; i+=8)  {
       r0 = *((__m64 *) linein1);
       r1 = *((__m64 *) linein2);
       r1 = _mm_max_pu8(r0,r1);
        *((__m64 *) lineout) = r1;

       linein1 += 8;
       linein2 += 8;
       lineout += 8;
   }
  }
#else
#ifdef __SSE2__
  template<>
  inline void t_LineArithSup1D<UINT8>(...) {
   int i;
   __m128i r0,r1;

   for(i=0 ; i<=size ; i+=16)  {
       r0 = _mm_load_si128((__m128i *) linein1);
       r1 = _mm_load_si128((__m128i *) linein2);
       r1 = _mm_max_epu8(r0,r1);
       _mm_store_si128((__m128i *) lineout,r1);

       linein1 += 16;
       linein2 += 16;
       lineout += 16;
```

Templates and Vectorization

Intrinsic Specializations

Sup_....cpp

#ifdef __SSE__

Sup_UINT16.cpp

#ifdef __SSE__

Sup_UINT8.cpp

```
#ifdef __SSE__
   template<>
   inline void t_LineArithSup1D<UINT8>(...) {
     int i;
     __m64 r0,r1;

     for(i=0 ; i<=size ; i+=8)  {
         r0 = *((__m64 *) linein1);
         r1 = *((__m64 *) linein2);
         r1 = _mm_max_pu8(r0,r1);
          *((__m64 *) lineout) = r1;

         linein1 += 8;
         linein2 += 8;
         lineout += 8;
     }
   }
#else
#ifdef __SSE2__
   template<>
   inline void t_LineArithSup1D<UINT8>(...) {
     int i;
     __m128i r0,r1;

     for(i=0 ; i<=size ; i+=16)  {
         r0 = _mm_load_si128((__m128i *) linein1);
         r1 = _mm_load_si128((__m128i *) linein2);
         r1 = _mm_max_epu8(r0,r1);
         _mm_store_si128((__m128i *) lineout,r1);

         linein1 += 16;
         linein2 += 16;
         lineout += 16;
```

With Auto-Vectorization

Sup.cpp

```
template <class T>
struct supLine : public binaryLineFunctionBase<T>
{
   typedef typename Image<T>::lineType lineType;
   inline void _exec(lineType lIn1, lineType lIn2, size_t size, lineType lOut)
  {
    for (size_t i=0;i<size;i++)
       lOut[i] = lIn1[i] > lIn2[i] ? lIn1[i] : lIn2[i];
  }
};
```

CMake

Cross-platform native makefiles and workspaces generator

- Allows to build the sources on almost any platform and with any compiler
- Facilitates cross-compilation
- Handles compilation options (external libs, optimizations, documentation, . . .)
- Handles wrapped languages and types

Current tested platforms:

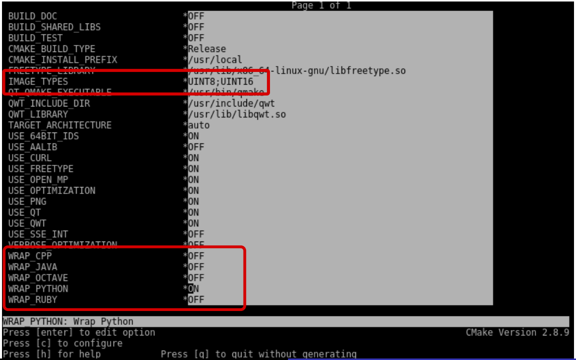
- Linux (32/64 bits) - GCC
- Windows (32/64 bits) - GCC, MSVC (without auto-vectorization)
- OSX - GCC, Clang
- Android

Swig

Current SMIL version allows to generate interfaces for **Java, Python, Octave and Perl**.

Generated code have the same function names and arguments as original C++ code.

The wrapped types and languages are defined via CMake.



Why Simple ?

Easy to use

- Simple framework (only 2D/3D images, no complex structures due to genericity)
- ~13,000 code lines (MorphM: ~130,000)
- Multi-platforms, multi-compilers (CMake)
- Serveral possible languages/interpreters with common native C++ code and the same function names and arguments (swig): python, java, octave, perl, ...
- Integrated viewer, events management
- Simple and short function names
- Intuitive approach:
    - functions overload
    - operators overload
- webstart version, ...

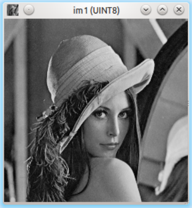
Examples of use

 # Load an image

```
im1 = Image()
im1 << "lena.png"
im1.show()
```

Equivalent to

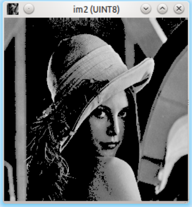
read("lena.png", im1)



 # Mask of values > 100

```
im2 = Image(im1)
im2 << ((im1>100) & im1)
im2.show()
```

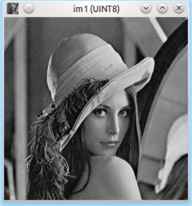
# Equivalent to
grt(im1, 100, im2)
inf(im1, im2, im2)

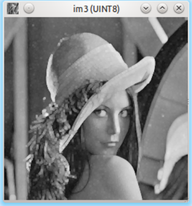


Examples of use

 # Sup of translations...

```
im3 = Image(im1)
sePts = ((0,0),(0,1),(1,1),(1,0),(-1,1),(-1,0),(-1,-1),(0,-1),(1,-1))
im3 << 0
for (dx,dy) in sePts:
    im3 |= trans(im1, dx, dy)
im3.show()
```

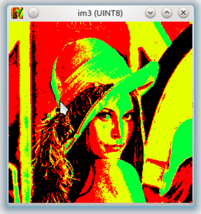




Examples of use

```
# Some thresholds

# Fixed threshold
threshold(im1, 100, 255, im2)
# Otsu (2 modalities)
threshold(im1, im2)
# Otsu (3 modalities)
otsuThreshold(im1, im3, 3)
im3.showLabel()
```



Why Simple ?

Effort to factorize as much as possible

- line functors
- image functors (standard operations, morphological operations, . . .)
- swig wrap definitions (simple swig macros to export functions/classes)
- . . .

⇒ Most complex/redundant/optimization operations are handled jointly
⇒ Contributors do not need much knowledge to add new features

Example: Dilation function

```
template<class t>
unaryMorphImageFunction<T, supLine<T> > dilateFunc();
```

⇒SIMD, Parallelization, specialized SE functions, . . .

1. Introduction
2. SMIL: General presentation
3. Optimizations
4. Benchmarks/Perspectives

Used optimizations

Software → Hardware

Line Processing

Hierarchical Queues

SE decomposition

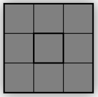
SIMD

Parallelization

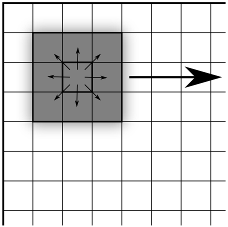
Binary Images

Line processing in morphological operations

The classical approach: Neighborhood iterator



Square
SE

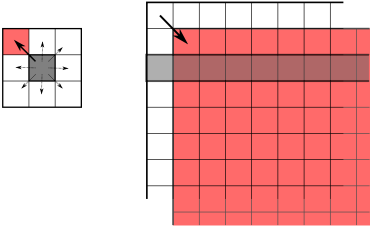


Pseudo-code (dilation):

```
for (j=0;j<height;j++)
  for (i=0;i<width;i++) {
    maxV=0;
    for (it in sePts) {
      outPix(i,j)=MAX(maxV, inPix(i+it.x,
j+it.y));
     }
     outPix(i,j)=maxV;
  }
```

Line processing in morphological operations

The line approach: Using image translations

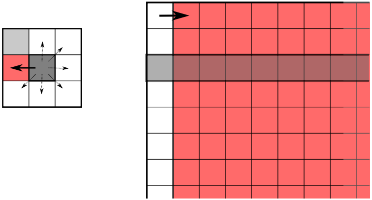


Pseudo-code (dilation):

```
for (j=0;j<height;j++) {
  fill(outLine(j),0);
  for (it in sePts) {
    tmpLine=translate(inLine(j+it.y),it.x);
    for (i=0;i<width;i++) {
      outLine(i)=MAX(outLine(i),tmpLine(i));
    }
  }
```

Line processing in morphological operations

The line approach: Using image translations



Pseudo-code (dilation):

```
for (j=0;j<height;j++) {
  fill(outLine(j),0);
  for (it in sePts) {
    tmpLine=translate(inLine(j+it.y),it.x);
    for (i=0;i<width;i++) {
      outLine(i)=MAX(outLine(i),tmpLine(i));
    }
  }
```

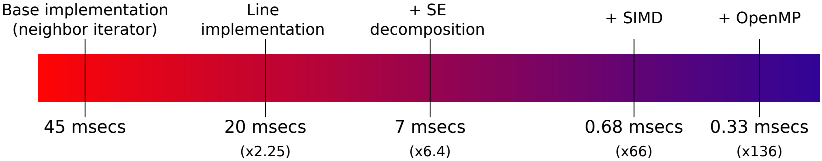
Line processing in morphological operations

Gains

- Speed (>x2)
- Allows some parallelization
- Allows to use SIMD

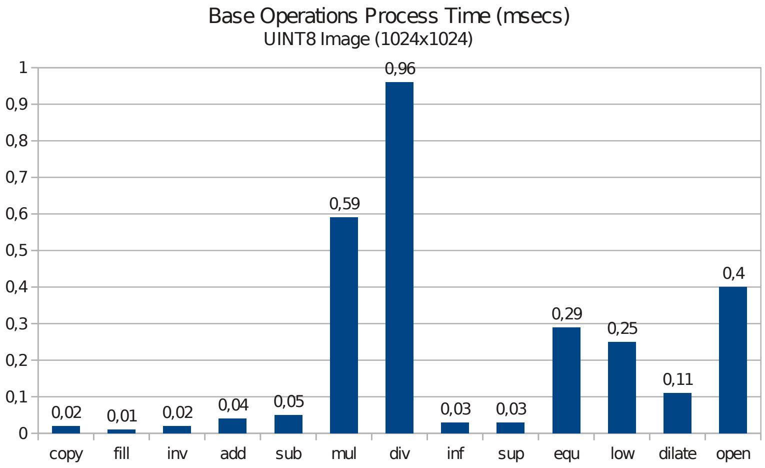
Example function: Morphological Dilation

Morphological dilation on 1024x1024 8bits image using a square structuring element:

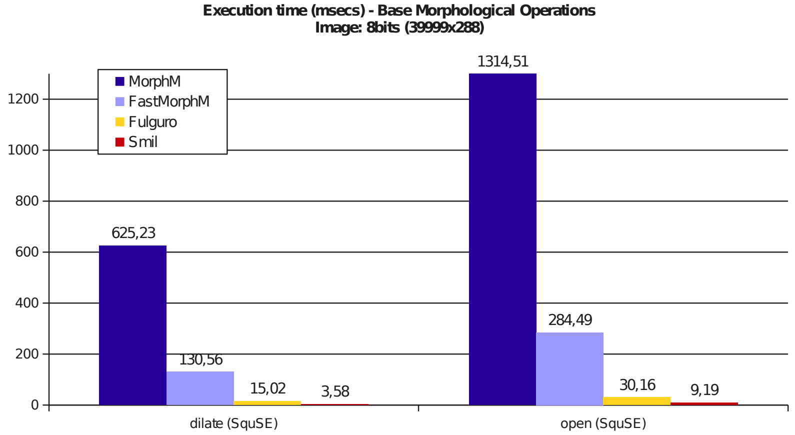


(Intel® Core™ i3 CPU M330 @2.13GHz, 2 cores, 4 threads)

Benchmarks

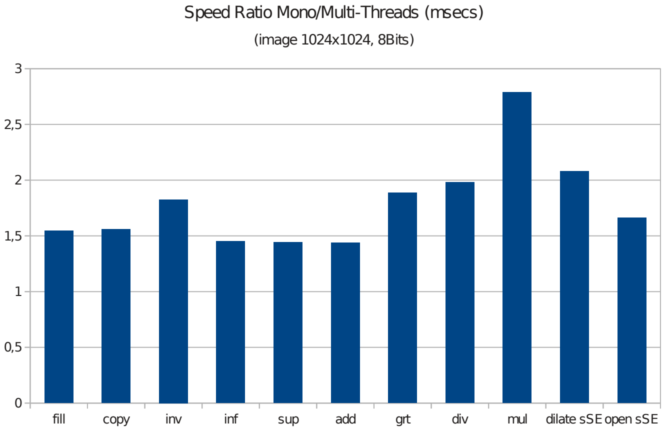

Base Operations Process Time (msecs)
UINT8 Image (1024x1024)
1
0,9
0,8
0,7
0,6
0,5
0,4
0,3
0,2
0,1
0
0,02
0,01
0,02
0,04
0,05
0,59
0,96
0,03
0,03
0,29
0,25
0,11
0,4
copy
fill
inv
add
sub
mul
div
inf
sup
equ
low
dilate
open

Benchmarks



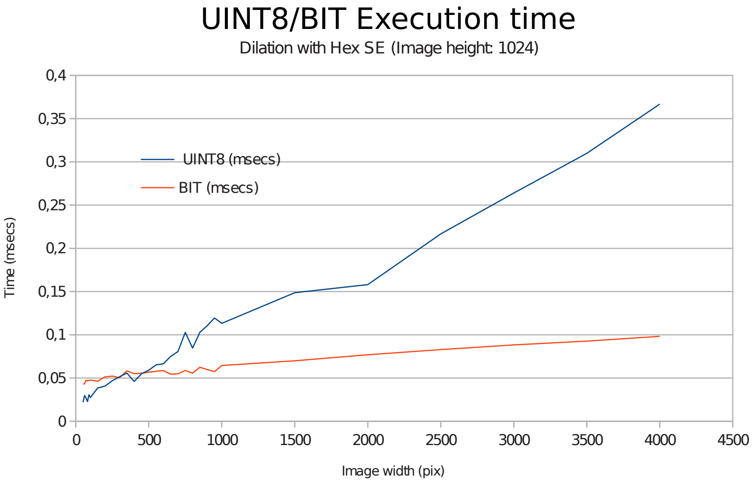
(Intel® Xeon® CPU E31245 @3,30GHz, 4 cores, 8 threads)

Benchmarks



(Intel® Core™ i3 CPU M330 @2.13GHz, 2 cores, 4 threads)

Benchmarks


UINT8/BIT Execution time
Dilation with Hex SE (Image height: 1024)
0,4
0,35
0,3
0,25
0,2
0,15
0,1
0,05
0
Time (msecs)
UINT8 (msecs)
BIT (msecs)
0
500
1000
1500
2000
2500
3000
3500
4000
4500
Image width (pix)

SLOC

SLOC	Directory	SLOC-by-Language (Sorted)
3440	Morpho	cpp=3440
3107	Base	cpp=2941,python=166
2023	Core	cpp=2023
1551	Gui	cpp=1551
1519	NSTypes	cpp=1519
749	IO	cpp=749
430	doc	python=430
297	CMake	python=297
222	Addons	cpp=222
51	test	cpp=31,java=20
0	top_dir	(none)

Totals grouped by language (dominant language first):

cpp:	12476 (93.18%)
python:	893 (6.67%)
java:	20 (0.15%)

Total Physical Source Lines of Code (SLOC) = 13,389
Development Effort Estimate, Person-Years (Person-Months) = 3.0
(Basic COCOMO model, Person-Months = 2.4 * (KSLOC**1.05))
Schedule Estimate, Years (Months) = 0.82 (9.82)
(Basic COCOMO model, Months = 2.5 * (person-months**0.38))
Estimated Average Number of Developers (Effort/Schedule) = 3.73
Total Estimated Cost to Develop = $ 411,839
(average salary = $56,286/year, overhead = 2.40).
SLOCCount, Copyright (C) 2001-2004 David A. Wheeler

Perspectives

Still much work to be done...

- More complex operations optimization:
  - Median, mean, ...
  - Labellization
  - ...
- Hierarchical queues optimization:
  - watershed
  - geodesic reconstruction
- GPU ?

Thank you...

SMIL web page:

http://cmm.ensmp.fr/~faessel/smil/

Online WebStart Version (requires Java):

http://cmm.ensmp.fr/~faessel/smil/doc/webstart_page.html