MathVisionC++ for Efficient Medical Image Processing in Mathematica

Using a Mathlink Caching Scheme

Erik Franken, Bart Janssen

Eindhoven University of Technology
Department of Biomedical Engineering

Mathematica User Conference 2008
June 20-24th 2008
Maastricht, The Netherlands
Outline

• Introduction
  – MathVisionTools
  – Why Create a C++ Library for Image Processing?
  – Problems with MathLink

• Mathlink Interface
  – Mathlink Caching
  – C++ Interface

• MathVisionC++ Library
  – Algorithms
  – Design

• Examples

• Conclusions
Introduction

Why use mathematica for (mathematical) image analysis?

• Convenient for fast translation of Mathematics (e.g. analytical formulas) to numerical implementation (e.g. on pixel/voxel data)

• Implementing simple image processing algorithms is relatively fast

• Useful built-in functionality for visualizing results, for instance `ListDensityPlot` (not useful anymore in version 6), `ListContourPlot`, `ListContourPlot3D` etc.
MathVisionTools

- MathVisionTools is a Mathematica library for image processing and analysis
- Including:
  - Differential Geometry (e.g. Gaussian derivatives for any order for N dimensions, Geometry driven diffusion)
  - Orientation analysis (e.g. Polar Fourier Transform, 2D Hankel Transform, G- convolutions, Stochastic Completion Kernels)
  - Visualization functions (e.g. for visualizing tensorial images)
  - Import / Export (e.g. of DICOM format)
- Optimized (fast) Mathematica code

Example: Differential Invariants

Expression for Rotation invariant T-junction detection:

\[ \frac{1}{(L_x^2 + L_y^2)^3} \left( -L_{xxy} L_y^5 + L_y^4 (2 L_{xy}^2 - L_x (L_{xxx} - 2 L_{xyy}) + L_{xx} L_{yy}) + \right. \]

\[ \left. L_x^4 (2 L_{xy}^2 - L_x L_{xyy} + L_{xx} L_{yy}) + L_x^2 L_y^2 (3 L_{xx}^2 - 8 L_{xy}^2 + L_x (-L_{xxx} + L_{xyy}) - 4 L_{xx} L_{yy} + 3 L_{yy}^2) + \right. \]

\[ \left. L_x L_y^3 (6 L_{xy} (L_{xx} - L_{yy}) + L_x (L_{xxy} - L_{yyy}) + L_y (5 L_{xy} (-L_{xx} + L_{yy}) + L_x (2 L_{xy} - L_{yyy})) \right) \]
Why Create a C++ Library?

Often large datasets and sophisticated algorithms

→ Severe requirements on memory and processing power

→ *Mathematica* can not meet these requirements
  - Slow (interpreted language)
  - Redundant memory consumption (can’t specify datatypes)

Our goal: take advantage of the speed and efficiency of a C++ program and the advantages of *mathematica* at the same time

→ So: write the real function in C++ and provide interfaces in *Mathematica* through *Mathlink*
Problems with Mathlink

1. Data transfers are required ML-functions in C++ program ↔ Mathematica kernel
   *Consumes a significant amount of time* when working with large datasets
2. The provided Mathlink C interface is inconvenient

Solutions:
1. Mathlink *caching* mechanism
2. A C++ `MathLinkIO` class
Outline

• Introduction
  – MathVisionTools
  – Why Create a C++ Library for Image Processing?
  – Problems with MathLink

• Mathlink Interface
  – Mathlink Caching
  – C++ Interface

• MathVisionC++ Library
  – Algorithms
  – Design

• Examples
• Conclusions
MathLink Caching Mechanism

Mathematica
ML function call without caching:
result = MLSomeFunction[data]
return: List[...]

MathLink channel

C++ (utilizing class MathLinkIO)
MLSomeFunction

cache table
MathLink Caching Mechanism

**Mathematica**
- ML function call without caching:
  - `result = MLSomeFunction[data]`
  - `return : List[...]`
- Caching data:
  - `mlData = MLCache[data]`
  - `return : MLCacheId[1]`

**C++ (utilizing class MathLinkIO)**
- `MLSomeFunction`
- Cache table

id=1 { ... }
MathLink Caching Mechanism

**Mathematica**

ML function call without caching:
```mathematica
result = MLSomeFunction[data]
return: List[...]
```

Caching data:
```mathematica
mldata = MLCache[data]
return: MLCacheId[1]
```

ML function on cached data:
```mathematica
result = MLSomeFunction[mldata]
return: List[...]
```

**C++ (utilizing class MathLinkIO)**

ML function call without caching:
```
MLSomeFunction
```

Caching data:
```
id=1 { ... }
```

ML function on cached data:
```
MLSomeFunction
```
MathLink Caching Mechanism

Mathematica

ML function call without caching:
result = MLSomeFunction[data]
return: List[...]

Caching data:
mldata = MLCache[data]
return: MLCacheId[1]

ML function on cached data:
result = MLSomeFunction[mldata]
return: List[...]

Cache the result of a ML function:
mlresult = MLCache[
  MLSomeFunction[mldata]]
return: MLCacheId[2]

C++ (utilizing class MathLinkIO)

ML function call without caching:
MLSomeFunction

Caching data:

ML function on cached data:

Cache the result of a ML function:

Cache table

id=1 { ... }

id=2 { ... }

id=3 { ... }

ML function call without caching:
MLSomeFunction
MathLink Caching Mechanism

Mathematica

ML function call without caching:
result = MLSomeFunction[data]
return: List[...]

Caching data:
mldata = MLCache[data]
return: MLCacheId[1]

ML function on cached data:
result = MLSomeFunction[mldata]
return: List[...]

Cache the result of a ML function:
mlresult = MLCache[
    MLSomeFunction[mldata]]
return: MLCacheId[2]

Retrieving cached data:
result = MLGet[mlresult]
return: List[...]

C++ (utilizing class MathLinkIO)

MLSome-Function

Cache table

id=1 { ... }

id=2 { ... }

MLSomeFunction

MLSomeFunction
Mathlink Caching Features

• All ML-functions can transparently handle real data and cached objects
• What type of data can we cache?
  → All arrays containing *numerical* data that Mathematica can store as *packed arrays*
• Additional mathematica functions for controlling the cache:
  - `MLRemove` and `MLRemoveAll` - to clear the cache
  - `MLCacheInfo` – to obtain information on the stored elements
  - `MLReplace[id, expr]` – to replace the data to which `id` (which should be an `MLCacheId`) points by `expr`. Useful to prevent “memory leak” in the cache when calling `MLCache` multiple times
Example run:

```
In[271]: << MathVisionCpp
In[272]: mlimg = MLCache[img]
Out[272]= MLCacheId[1]
In[273]: mlkhn = MLCache[krn]
Out[273]= MLCacheId[2]
In[274]: MLCacheInfo[] // MatrixForm
Out[274]/MatrixForm=
   {MLCacheId[1] {16, 16} {List, List} RealArray}
   {MLCacheId[2] {5, 6} {List, List} RealArray}
In[275]: mresult = MLCache[MLListConvolve[mlkhn, mlimg, krcnt, BOUNDARYCYCLIC, CONVOLVEDIRECT]]
   MathVisionCpp::invalidparams: Invalid parameter types used. Function aborted.
Out[275]= MLInvalidArgumentNumber[2]
In[276]: mresult = MLCache[MLListConvolve[mlkhn, mlimg, krcnt, BOUNDARYCYCLIC, CONVOLVEDIRECT]]
Out[276]= MLCacheId[3]
In[277]: result = MLGet[mresult]; Dimensions[result]
Out[277]= {16, 16}
In[278]: MLRemoveAll[]
In[279]: MLCacheInfo[]
Out[279]= {}
In[280]: mresult = MLCache[MLListConvolve[mlkhn, mlimg, krcnt, BOUNDARYCYCLIC, CONVOLVEDIRECT]]
   MathVisionCpp::mlcacheerror: Mathlink cache error.
Out[280]= MLInvalidArgumentNumber[1]
```
Taking Advantage of Caching

- Using a large data set in multiple ML-function calls

```plaintext
mldata = MLCache[data];
result1 = MLFunction1[mldata];
result2 = MLFunction2[mldata];
result3 = MLFunction3[mldata]; ...
```

- Using the result of a ML-function in a subsequent ML-function call

```plaintext
mlresult1 = MLFunction1[data]//MLCache;
mlresult2 = MLFunction2[mlresult1]//MLCache;
finalresult = MLFunction3[mlresult2];
```
MathLinkIO Class in C++

C++ interface to handle Mathlink communication, featuring

• Get(...) and Return(...) functions for many different standard data types + data types defined in the MathVisionC++ library. These functions transparently handle caching and data type conversion

• Error handling: in case of mathlink errors supplied in ML-function call, an MLErrorType object is thrown; catch block sends an error message to Mathematica

• In case of wrong argument types (e.g. wrong dimensionality of list), we use the same error handling scheme. This is far more user-friendly then using Mathematica pattern-testing on the arguments (we get an error message in Mathematica instead of undesired continuation of symbolic calculations)
Coding Example with MathLinkIO

We developed mprepprep, a small tool that automatically generates mprep files from our C++ code.

```cpp
// squareimage.cpp
#include <mathvisioncpp/datatypes/multiarray.hpp>
#include <mathvisioncpp/inputoutput/mathlink_io/mathlink_io.hpp>

using namespace MathVisionCpp;

/* MPREP
:Mathematica: MLSquareImage[image_]
*/
void MLSquareImage()
{
  IO::MathLinkIO mma;
  try {
    DataTypes::MultiArray<double,2> image;
    mma.Get(image, true); // "true" is to indicate for in-place-processing
    image *= image;
    mma.Return(image);
  } catch (IO::MLErrorType& i)
  { mma.Abort(i); }
}
```

Automatically generated mprep file

```
:Begin: MLSquareImage
:Function: MLSquareImage[image_]
:Pattern: {image}
:Arguments: {Manual}
:ReturnType: Manual
:End:
```
Outline

• Introduction
  – MathVisionTools
  – Why Create a C++ Library for Image Processing?
  – Problems with MathLink

• Mathlink Interface
  – Mathlink Caching
  – C++ Interface

• MathVisionC++ Library
  – Algorithms
  – Design

• Examples
• Conclusions
Considerations on MathVisionC++

- Support Any-D data processing (not only 1-2-3-D)
- Efficiency and code reusability
- Small effort on user interfaces (only command-line + mathematica interface)
- Should provide alternative to Mathematica for the *larger, more complex* algorithms and experiments
- Students with a bit of C/C++ experience should be able to write their code within the library framework
- Platform independent
Most Important Algorithms

Any-D
- FIR filters, i.e. ListConvolve + ListCorrelate (spatial+Fourier); data types and convolution algebra are template parameters
- Separable FIR filters
- Inner-product taker
- Fourier transform (i.e., wrappers to FFTW functions)
- Gaussian derivative (Jet) and GD (Jet) At
- Unser’s spline interpolation

Specific-D
- Scale space interest points
- Non-linear diffusion
- Orientation scores (OS)
  - Orientation score transformation
  - G-convolutions (steerable + non-steerable)
  - OS Gaussian derivative Jet
  - OS non-linear diffusion scheme
Library Structure

Directory + namespace (first level only) structure:

- basics
- datatypes
- listfunctions
- inputoutput
  - mathlink_io
  - file_io
- interpolation
- convolve
- specialfunctions
- solvers
- transforms
- reconstruction
- morphology
- scalespace
  - lineardiffusion
  - interestpoints
  - gaussian derivatives
- orientationscores
  - ostransforms
  - osmorphology
  - osdiffusion
  - osderivatives
  - gconvolve
- heisenbergscores
  - heisenbergdiffusion
  - heisenberggtransforms
- orientationscores3d
Library Dependencies

- Dependencies on other libraries:
  - Boost
  - FFTW
  - MathLink
- Work in progress: using BLAS/LAPACK to increase speed
- Furthermore we use
  - Cmake cross-platform build environment
  - Doxygen for documentation and tutorials
  - SVN for version control
Core datatypes: Vect and MultiArray

- **MultiArray<T, Rank>** is a Rank-dimensional datatype with elements of type T. For example
  
  ```
  MultiArray<double, 3> foo;
  MultiArray<Matrix<float,2,2>, 2> bar;
  ```
  - Provides “auto-allocation”, operators (e.g. *+, += etc), sub-array iterators etcetera.

- **Vect<T, Len>** is a Len-dimensional vector of type T.
  ```
  Vect<int,3> myIntVec;
  Vect<double,2> myDoubleVec;
  ```

- Rank of MultiArray and Len of Vect are template parameters, so values are known during compilation → Allows for more optimization by compiler.
Mathematica-inspired functions

- Mathematica-like operations in vectors, e.g.
  
  ```
  Append(myIntVec, 10), Prepend(myIntVec, 10)
  Take<0,2>(myIntVec), Join(myIntVec, myOtherIntVec)
  ```

- Mathematica-like operations on MultiArray, e.g.
  
  ```
  Map_<0>(_1 > thresholdValue, thresholdedImage, niceImage);
  ```

  Compare:
  ```
  thresholdedImage =
  Map[If[#1>thresholdvalue,1,0]&, niceImage, {2}];
  ```

  (Note: “level-spec” convention is other way around)

- Also implemented parallel versions
  (e.g. ParallelMap)
Algorithm Interfaces

  Example:
  
  ```
  { 
    GaussianDerivativeJet<T,Rank> MyGD(scale, maxDerivativeOrder, imageDimensions, boundaryConditions, convolutionMethod); 
    MyGD.Calculate(result1, input1, 1); 
    MyGD.Calculate(result2, input2, 1); 
  } // C++ will now call MyGD’s destructor
  ```

- **Function interface**: provides kind of shortcut to most important algorithms. Example:

  ```
  ListConvolve(result, image, kernel, 
              kernelCenter, boundaryCondition, convolutionMethod);
  ```
Outline

• Introduction
  – MathVisionTools
  – Why Create a C++ Library for Image Processing?
  – Problems with MathLink

• Mathlink Interface
  – Mathlink Caching
  – C++ Interface

• MathVisionC++ Library
  – Algorithms
  – Design

• Examples

• Conclusions
Examples

• All examples were run on the bigmath2 machine:
  – 4 Dualcore Opteron 2.2Ghz CPUs
  – 64GB of ram
  – Linux
  – Mathematica 6.02
  – Gcc version 4.1.2

• The examples will demonstrate:
  – Speed difference between MathVisionC++ and Mathematica / MathVisionTools
  – Benefits of using Mathlink caching
Example: Interpolation

1. Mathematica:
   \[ \text{imgintMma} = \text{ListInterpolation}[\text{img}, \text{InterpolationOrder} \rightarrow 1] \]
   \[ \text{results} = \text{imgintMma}@@\{\{x_1, y_1\}, \{x_2, y_2\}\} \]

2. MathVisionC++ via Mathlink Interface:
   \[ \text{imgintCpp} = \text{MLNewBSplineInterpolator}[\text{mlimg}, 1, \text{MLBOUNDARYREFLECTIVE}] \]
   \[ \text{results} = \text{MLBSplineInterpolatedValue}[\text{imgintCpp}, \{\{x_1, y_1\}, \{x_2, y_2\}\}] \]

<table>
<thead>
<tr>
<th></th>
<th>Total calculation time (including transfer)</th>
<th>Total transfer time</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. Mathematica</td>
<td>8.8 s</td>
<td>-</td>
</tr>
<tr>
<td>2. MathVisionC++</td>
<td>2.5 s</td>
<td>0.8 s</td>
</tr>
</tbody>
</table>

Transfer time includes transfer of image, list of requested interpolation coordinates and interpolation results (B-Spline coefficients are automatically \textit{cached}; no need to transfer)
Example: GaussianDerivativeAt
Optic flow algorithm using multiscale toppoints

- Uses toppoints in scale space of an image sequence to estimate optic flow.
- Algorithm requires many GaussianDerivativeAt on the same 3D (2D+time) image: 19 calls for each toppoint.
- GaussianDerivativeAt: input=3D image, output = scalar value.

Timing results in seconds on bigmath2, single-threaded, Yosemite sequence (316 pixels x 251 pixels x 15 frames), 819 Toppoints, 819x19 = 15561 x GaussianDerivativeAt

<table>
<thead>
<tr>
<th></th>
<th>Time per derivative</th>
<th>Total time (with transfer)</th>
<th>Total transfer time</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mathematica (MathVisionTools)</td>
<td>0.048 s</td>
<td>235 s</td>
<td>-</td>
</tr>
<tr>
<td>MathVisionC++ without any caching</td>
<td>0.4 s</td>
<td>6224 s</td>
<td>6212 s</td>
</tr>
<tr>
<td>MathVisionC++ with caching</td>
<td>0.00079 s</td>
<td>12.7 s</td>
<td>0.4 s</td>
</tr>
</tbody>
</table>
Example: Gradient of a 3D Image

Mathematica:

\begin{verbatim}
tx = 20.; ty = 10.; tz = 16.;
SetOptions[GaussianDerivative, Method -> Convolve];
Timing[
  gradient = {GaussianDerivative[{tx, 1}, {ty, 0}, {tz, 0}][img],
               GaussianDerivative[{tx, 0}, {ty, 1}, {tz, 0}][img],
               GaussianDerivative[{tx, 0}, {ty, 0}, {tz, 1}][img]};
  gradmag = \sqrt{(\#1^2 + \#2^2 + \#3^2)} \& @@ gradient;
]
\end{verbatim}

MathVisionC++ (Full C++, no Mathlink in this example):

\begin{verbatim}
_StartTiming();

MultiArray<double, 4> gradient;
CalculateGaussianDerivativeJet(gradient, img, Vect3(20., 10., 16.), 1, 1,
  BoundaryConditionVector<3>{Cyclic}, ConvDirect);

MultiArray<double, 3> gradmag;
Map_<1>(Norm<2>(), gradmag, gradient);

_StopTiming();
\end{verbatim}

Timing results on bigmath2, image size 171x100x195, all single-threaded

<table>
<thead>
<tr>
<th>Method</th>
<th>Mathematica</th>
<th>C++ (gcc4)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Convolve</td>
<td>13, 6 s</td>
<td>2,9 s</td>
</tr>
<tr>
<td>Fourier</td>
<td>34,1 s</td>
<td>3,3 s</td>
</tr>
</tbody>
</table>
Outline

• Introduction
  – MathVisionTools
  – Why Create a C++ Library for Image Processing?
  – Problems with MathLink

• Mathlink Interface
  – Mathlink Caching
  – C++ Interface

• MathVisionC++ Library
  – Algorithms
  – Design

• Examples

• Conclusions
Conclusion

• Proposed a Mathlink *caching* scheme to limit the amount of Mathematica ↔ C++ data transfers
• Introduced MathVisionCpp: a C++ library for image processing
• Difference from other C(++) libraries: *Any-D* & more *mathematically-oriented* functions.
• Provide a full interface to Mathematica, to take advantage of strengths of both Mathematica and C++
• Our C++ implementations prove to be faster
• Caching mechanism allows for a large Mathlink transfer overhead reduction
• Under development, but already a lot of functionality
Acknowledgements

• Roel Jordans (Student assistant documenting and improving MathVisionC++)

• Pieter van Dorst (GaussianDerivativeAt example)

• Markus van Almsick (MathVisionTools development)

• Bart ter Haar Romeny (funding)