Refactoring and Improving an Optical Flow Calculation Algorithm
Bachelor-Thesis

Client & Mentor	Dr. Ralf Kapulla
Student	Dimitri Nüscheler
Coach	Prof. Dr. Christoph Stamm
Version	1.1
Last revision	25.3.2011

Revisions

Revision	Description	Author
0.5	Most important aspects of design and implementation added	Dimitri Nüscheler
0.7	Complete document structure, corrections and completions	Dimitri Nüscheler
0.9	Content nearly complete	Dimitri Nüscheler
1.0	Content is complete, ready for reviewing	Dimitri Nüscheler
1.1	Complete	Dimitri Nüscheler

1 Abstract

This project introduces an object oriented design, parallelization and enhanced batch processing to the Optical Flow Calculation Algorithm that has been developed at the Paul Scherrer Institute.

The problem of determining the optical movement vector field accurately in a discrete image signal requires an iterative process that starts by scaling down the problem size to later advance to higher resoluted signal data that makes use of results of previous iterations.

By introducing a refactoring to object oriented standards geared towards declarative design the algorithm should allow for more flexible configuration of the actual problem, makes adding and replacing modules easier and makes the iterative concept of the algorithm easier to understand. It also introduces the observer pattern for separating the monitoring aspect from the actual problem.

Based on the refactored solution configuration profiles, persistence and batch processing is implemented. By using MATLABs parfor-construct and by interfacing with NVIDIA's CUDA Toolkit two distinct parallelization approaches are successfully demonstrated.

Combining parallelization, monitoring and batch processing the framework allows for setting up and running large-scale tests in a short time.

Finally the test results already provided and the documented use cases give an idea of future development.

Table of contents

Revisions....2
1 Abstract....3
2 Introduction....5
  2.1 Purpose of this document....5
  2.2 Document structure....5
3 Initial position....6
  3.1 Illustration of the problem....7
  3.2 Illustration of the iterative process....8
  3.3 Sequence diagram of the existing code....9
  3.4 Defiencies....10
4 Project goals....10
  4.1 Non-functional requirements....11
  4.2 Functional requirements....12
5 Planning & Controlling....13
6 Quality Assurance....14
7 Design....15
  7.1 Description of the algorithm using object oriented terms....15
  7.2 Adressing defiencies....17
  7.3 Monitoring....18
  7.4 Design rationale....18
  7.5 Platform....19
8 Implementation....21
  8.1 Layer model – Design overview....21
  8.2 Class diagram (base, implementation and monitors)....22
  8.3 User interface facade....24
  8.4 Parallelization....31
  8.5 CPU-Parallelization....31
  8.6 GPU-Parallelization....32
  8.7 Walkthrough & Use Cases....33
  8.8 Changelog....40
  8.9 Implemented requirements....41
  8.10 Remaining issues....42
9 Testing....43
  9.1 Performance....43
  9.2 Accuracy....50
10 Reflection....54
11 Outlook....55
12 Glossary....56

2 Introduction

2.1 Purpose of this document

This document provides an overview of the Refactored Optical Flow Algorithm. „Refactored“ as it is based on an existing project developed at the PSI.

The document describes the state of the existing solution at the start of this project, description of the process, the design decisions, the actual refactored implementation, quality assurance, testing and reflection as well as an outlook on future development of the project.

2.2 Document structure

The document is divided into 12 chapters. Introductional chapters, chapters related to the software development process and finishing chapters.

2.2.1 Introductional

Chapters: Abstract, Introduction (this chapter) and Initial position.

These chapters explain what the project is about and the motivation.

2.2.2 Software development

Chapters: Project goals, Planning & Controlling, Quality Assurance, Design, Implementation and Testing

These chapters are related to the software development process (Requirements Engineering, Planning, Design, Implementation & Testing). Product and process are described.

2.2.3 Finishing chapters

Chapters: Reflection and Outlook

These chapters reflect on the project and propose further development

3 Initial position

Ralf Kapulla is working on an algorithm for optical flow calculation at the PSI. A complete solution written in MATLAB is existent and is in process of development.

The algorithm aims at determining a fine-grained vector field that describes the optical flow between 2 images.

Determination of the optical flow potentially has a wide range of application areas such as observation of fluid dynamics, robotics or similar.

Unlike other algorithms such as motion compensation algorithms (MPEG) which generate one vector per image area it tempts to find a precise optical flow field with a resolution of one floating point vector per pixel.

The algorithm is iterative, which means that it solves the problem for smaller and blurrier versions of the original images and then reuses the result on higher resoluted and sharper images.

The actual result is determined using a linear equation which is constructed based on two conditions[1]:

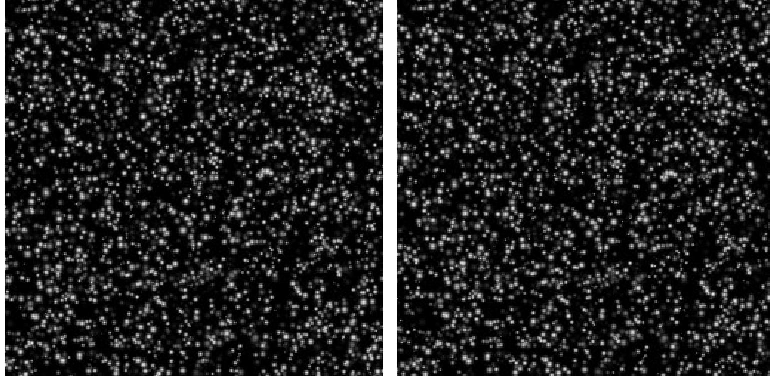
- Energy: Minimize intensity change or gradient when following a pixels flow vector from the first image to the second.
- Smoothness: The resulting flow field should be smooth.

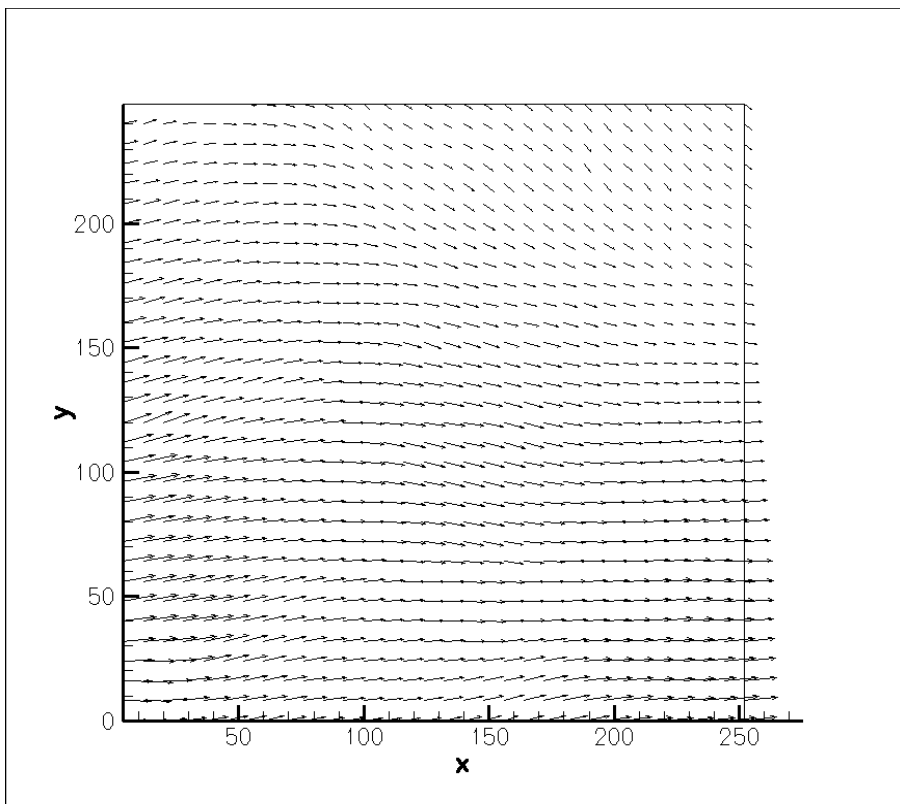
If an initial guess of the flow field is available, which is the case after the first iteration, the image pair is moved (warped) together along the flow field before the linear equation is constructed from the image pair. The resulting flow field describes the remaining difference between the 2 images that the coarser grained calculation in the previous iteration wasn't capable of determining.

The solution provides monitoring capabilities which creates a history during execution so that the user can look for weak spots.

3.1 Illustration of the problem

Applying the optical flow calculation to this image pair with a resolution of 256x256x2





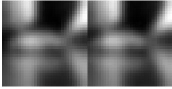
Optical flow, VSJ Pair 1

The result is a vector field with the same resolution as the input signal.

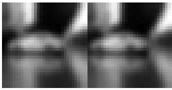
The vector displayed here is not the final result. It is taken at the after the first **pyramid level** finished and thus has only a resolution of 32x32.

3.2 Illustration of the iterative process

Pyramid Level 4, Scale Level 1

 For a small and blurry version of the image pair the optical flow is calculated

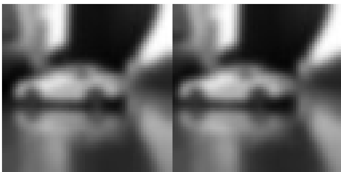
Pyramid Level 4, Scale Level 2

 The algorithm proceeds based on the previous result with a sharper version of the image

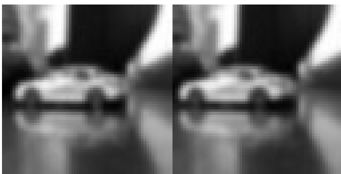
Pyramid Level 4, Scale Level 3

Finally the smallest pyramid level executed with an even sharper version.

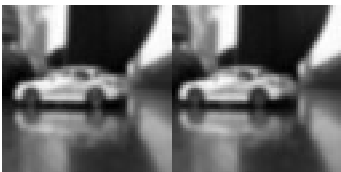
Pyramid Level 3, Scale Level 1

 The result of the previous pyramid level is scaled to a higher resolution and used on an equally resoluted image pair.

Pyramid Level 3, Scale Level 2

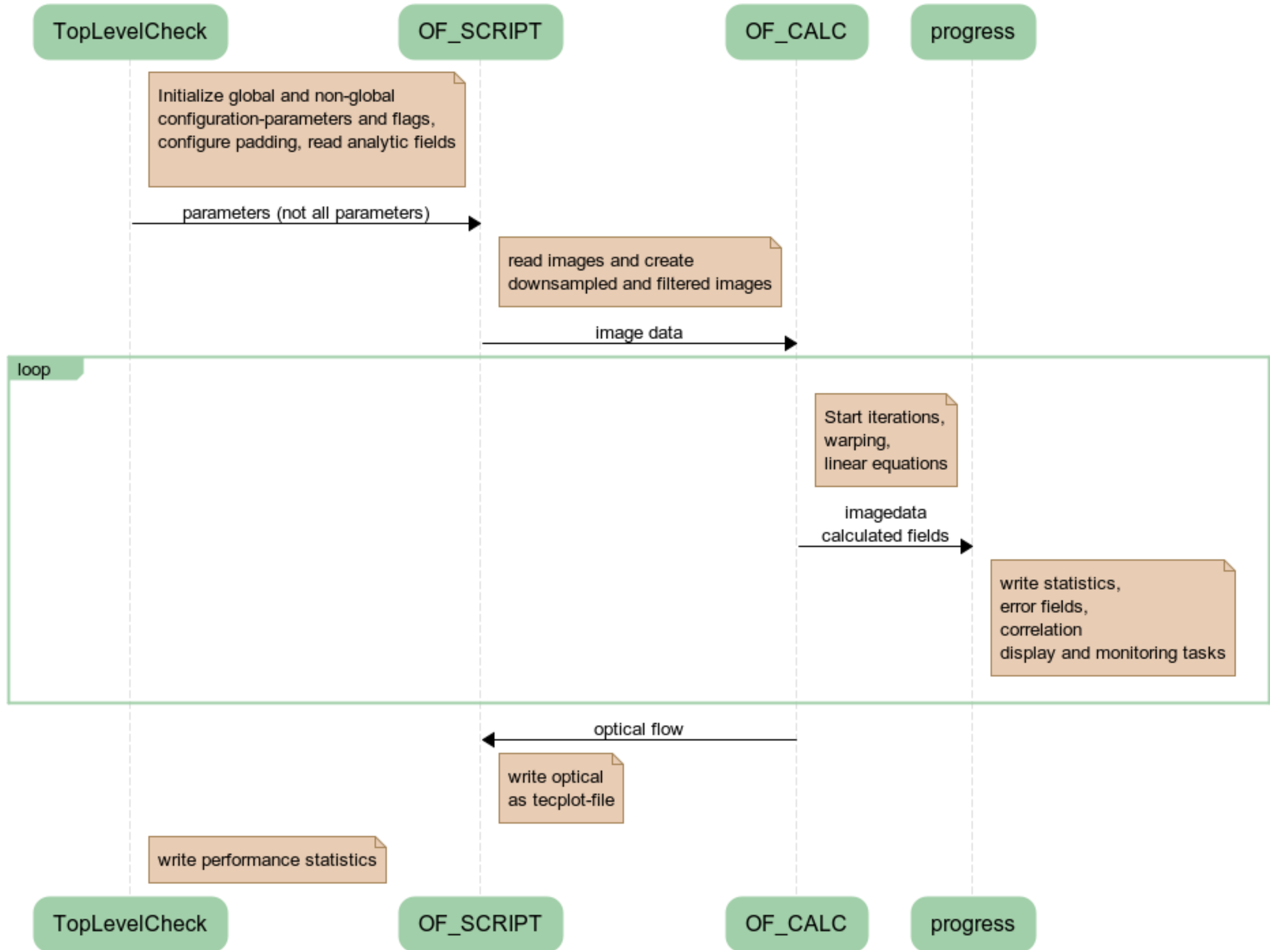


Pyramid Level 3, Scale Level 3



When the algorithm reaches the bottom pyramid level (Level 1), the result is returned.

3.3 Sequence diagram of the existing code



TopLevelCheck is responsible for defining parameters and global variables and writing time statistics.

OPTICAL_FLOW_SCRIPT is responsible for calling the image reading routines and filter functions and writing the flow fields into a format that can be read by tecplot.

OPTICAL_FLOW_CALCULATION does the iterations and the actual calculation

progress is responsible for monitoring the iterative process.

3.4 Defiencies

The defiences of the current solution is one of the motivations for this project. Below these defiencies are described.

Many features have been added to the software, thus the software has grown in complexity. However, as of now no object oriented principles were applied to the solution, making it a difficult task to extend and reuse the implementation.

The following deficiencis were detected during examination of the implementation:

Refactoring issues

- Long parameter lists
- Mixed use of parameters and global variables
- Strong coupling between modules of the algorithm
- Strong coupling between different aspects such as problem solving and monitoring
- Problem solving code and monitoring share persistence-responsibilities

In addition the software currently lacks parallelization and usability to timely allow large amounts of signal data to be processed.

4 Project goals

The refactoring issues mentioned above build the base for the project. After the refactoring is complete, it is possible to address further issues such as parallelization, batch processing and user interfaces.

The requirements to the project are listed below and divided into non-functional requirements, which address the refactoring issues, parallelization and usability, and functional requirements, which also includes the way certain problems are solved, because future users are going to inspect the source-code and expect certain behaviour in certain places.

To some extents, recommendations of the client already exist on how to implement these requirements.

Each requirement receives an ID, dependencies and a priority level.

4.1 Non-functional requirements

ID	Dep.	Priority	Name	Description	Recommendations
C01	-	High	Extendability and Modifiability	It should be easy to replace parts of the algorithm such as the solver, the pyramid and scale level iteration routines or the warping algorithm.	Use better modularization
C02	C01	Medium	Performance	-	Use parallelization. Where advantageous, externalize code to Java or C++.
C03	-	Medium	Usability	Decisions related to usability should take into consideration that the project does not aim at being an industry product, but at being a research project used by professionals.	
C04	-	High	Accuracy	The error of the calculated results should be minimal. A measurement is to be established and results reported.	

4.2 Functional requirements

Since this project is a research project the functional requirements do not only cover solving the optical flow problem, but also monitoring the algorithm in order to find weak spots.

ID	Dep.	Priority	Name	Description	Recommendations
C05	C01	High	Tecplot Monitor	In order to maintain compatibility with Tecplot, the refactored solution should be able to write tecplot files.	
C06	C01	High	Performance Monitor	How the algorithm performs should be written into a log file. The log file should contain how each module performs.	The performance profiler should log based on the call-hierarchy.
C07	C09	High	Batch processing	It should be possible to direct the algorithm to process multiple image pairs at once.	
C08	C01	Low	Cross correlation	An initial guess for the complete algorithm should be constructed using a cross correlation function	
C09	C01	High	Single Run Profiles	Using configuration files it should be possible to pass parameters to the algorithm. It must be possible to vary parameters along scale and pyramid levels.	
C10	C09	High	Batch Profiles	Using configuration files it should be possible to configure a batch process and it should be possible to persist a batch configuration for reuse. It should be possible to vary parameters along batch iterations.	
C11	C01	Medium	Read image files	Compatibility with Bitmap, Tiff and DaVis (im7) files	Use pivmat to access im7
C12	C09	Low	Graphical User Interface	It should be possible to configure the algorithms parameters using a user interface.	

5 Planning & Controlling

At the beginning of the project requirements were identified. The most important requirement were the refactoring issues which had to be implemented first.

During implementation the next steps and further requirements were regularly discussed and documented. Since I am working on the project alone the implementation plan can be derived from the requirements listing by sorting by dependence and priority. Since there aren't any „synchronization“ deadlines there is no direct need to do effort estimation. Instead a roughly guessed release plan is found below:

RIDs	Name	Finish date
C01, C03, C04	Extendability and Modifiability, Usability, Accuracy (Refactoring)	12.2010
C05	Tecplot Monitor	CW 7, 2011
C06	Performance Monitor	CW 7, 2011
C09	Single run Profiles	CW 7, 2011
C07	Batch processing	CW 8, 2011
C10	Batch profiles	CW 8, 2011
C02	Performance	CW 9, 2011
C11	Read image files	CW 9, 2011
C08	Cross correlation	CW 10, 2011
C12	Graphic User Interface	CW 10, 2011
	Complete documentation	CW 11, 2011
	Handover	CW 12, 2011

6 Quality Assurance

In order to assure quality of the refactored solutions the following measurements have been applied:

Testing

Performance and Accuracy of the refactored solution have to be tested. Both items are covered in the Testing section.

Frequent meetings with the client

The client is informed often about the current state of the project and needed to gather knowledge required to understand the area.

Prototyping

As often as possible the client is enabled to try out prototypes.

Explanatory documentation

The client is new to object oriented programming. Thus it is important to explain the object oriented programming paradigm and to document clearly.

Tool-drivenness

Subversion is used in order to ensure data security and to resolve regressions.

7 Design

When designing the refactored solution of the existing algorithm the following questions came up:

- How can the algorithm be described using object oriented terms?
- How are the defiencies mentioned in the initial position best adressed using object oriented measures?
- Which parts are problem solving code and which parts are monitoring code and how do you decouple them?
- If a new design is found, is it reasonable?

7.1 Description of the algorithm using object oriented terms

Observing the algorithm the following characteristics show up:

It is **iterative**, which means that it solves the optical flow problem for a preprocessed input and then reuses the refined output as an approximation for the next iteration with other preprocessing parameters. It solves the optical flow problem several times.

There are 2 types of data used during the algorithm. The first type is **signal data**. The user passes an input signal data to the algorithm (an image pair), the signal is modified internally and transferred to other components of the algorithm. During the process the algorithm creates output signal data (the optical flow) which is also passed between components of the algorithm. In the end the user receives the final output signal data, the result of the calculation.

The second type of data are **user parameters**. The algorithm allows for modification in many places. These parameters do not change during algorithm execution and they are not derived from the signal data.

The yet undiscussed part of the algorithm is the component that calculates the optical flow, the actual **core calculation**. From an object oriented point of view it is the most trivial. Conceptually it can best be described as a function that receives an image pair, an approximate optical flow solution and returns a new (more precise) optical flow solution. However, this component is very delicate to what input signal you pass to it (it requires the mentioned preprocessing) and returns useless results otherwise.

In terms of object-oriented programming there is only one clear contract which is used throughout the algorithm. The contract of components solving the optical flow is to **provide a function that takes two images plus eventually an initial solution and returns the optical flow between those images**. We call this contract **OFCProcessor**.

This contract should be implemented by the core calculation routine as well as the algorithm as a whole.

The algorithm as a whole is iterative. Thus we can describe the algorithm as an iterator, which implements **OFCProcessor,** we introduce **OFCIterator**.

How an **OFCIterator** works

1. Takes an image pair
2. Does some processing on the original images
3. passes processed images (and initial solution if existent) to another **OFCProcessor** object
4. refines the result if necessary
5. if the iteration counter reaches the final limit:

    return the result

    otherwise: go to step 2

The original algorithm applies 2 distinct kinds of image preprocessing at each iteration:

- Coarsing (or pyramid level generation): Downsample the original images.
- Blurring (or scale level generation): Apply a gaussian filter.

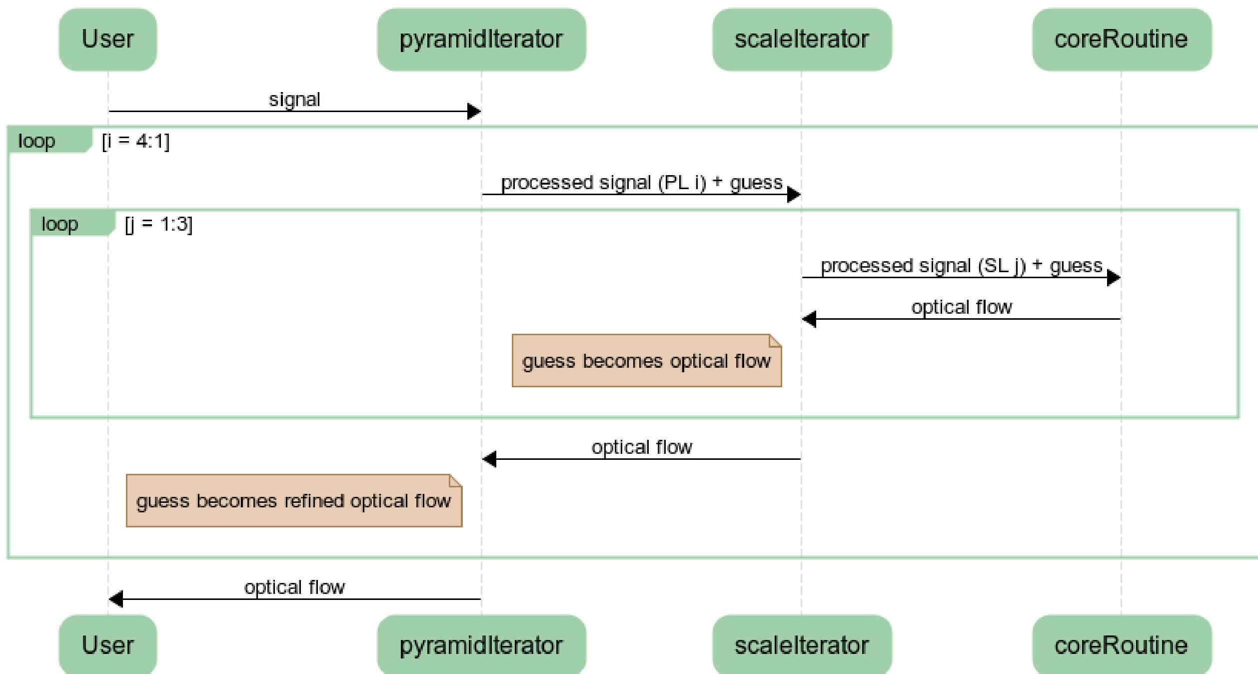
In the original algorithm both operations happen iteratively within 2 nested for-loops.

The coarsing occurs in the outer loop, the blurring in the inner loop.

In order to implement this behaviour we use an **outer** and an **inner** OFCIterator.

Step 3 of the outer OFCIterator delegates the signal to the inner iterator.

Step 3 of the inner OFCIterator delegates the signal to the core calculation.



Algorithm sequence diagram

pyramidIterator, scaleIterator and coreRoutine are implementations of OFCProcessor, depicting a chain.

pyramidIterator and scaleIterator are both instances of OFCIterator with distinct image processing routines applied.

7.2 Adressing defiencies

In the previous chapter we mentioned **user parameters**.

In the original algorithm they are randomly declared global, passed to functions or stored into structs.

This way of using parameters cause function declarations to become large and components are strongly coupled together.

In order to simplify parametrization of the algorithm and decouple components from each other, we decide to use a uniform way of defining them, which doesn't conflict with the previously defined contracts:

User parameters are stored as properties of related objects.

An object is related to a parameter, if it or its subroutines make use of that parameter.

7.3 Monitoring

In the existing solution monitoring is directly embedded into the algorithm. The same antipatterns such as heavy use of globals, long parameter lists and lack of clear contract exist.

Since monitoring code is code that runs during algorithm execution but doesn't have any influence on the result (has observer status only) it should be decoupled using the **observer pattern**.

However, the Observer-pattern doesn't imply observer status, see the Outlook section on how monitors could become part of the algorithm.

7.4 Design rationale

Now that we introduced several changes to the design of the optical flow calculation the question remains if these changes are reasonable and useful to the client.

There are some indicators that fortify our conclusions.

- OFCProcessor is a very simple and declarative contract.
- Declarative design[2] is better geared towards mathematical formalism than imperative design, which is desirable.
- The modularization doesn't aim at collecting every function into a class. The design affects the coarser design of the solution. There are no new concepts for finegrained datatypes required, because the existing MATLAB data structures are derived directly from mathematical concepts and thus are already in a desired state.

But there are also some weak spots:

- Probably, object oriented programming is rarely used for implementation at the Paul Scherrer Insitute.
- Instead of using an iterator the problem can also be formulated recursively where the result of the bottom level pyramid is described using the result of a higher level pyramid level. This approach is closer to the functional programming paradigm[3].

These weak spots are addressed using a user interface facade which hides the object oriented complexity and by leaving the options open to redefine the problem recursively.

The transition from procedural to functional code is safer by placing an object-oriented step in between anyway.

7.5 Platform

At the beginning the question came up about which programming language to use in order to implement the refactored solution. The languages that came into consideration were MATLAB, Java and C++. The existing code is written in MATLAB already.

In order to evaluate the best choice the languages had to be compared bearing in mind their capabilities for object oriented programming and parallelization.

MATLAB	**Java**	**C++**
- Type safety only at runtime (dynamic typing) (**Disadvantage:** Makes object oriented programming more difficult, because OOP is based on strong indirection so that without static typing the programmer eventually tends to lose the overview) (-) **Advantage:** Minimal syntactic overhead. Allows for specifying function declarations and interfaces before defining the type) (+) - Efficient matrix operations exist (+) - Multiple inheritance (+) - Restricted parallelization (-) - Proprietary (-) - Well known in the area (+) - Existing code is MATLAB (+) - Eventually allows to embed CUDA (+)	-Static typing (+/-, see MATLAB) - Assuming better performance than MATLAB, because of static typing and other checks that are already made at compile time (+) - Matrix operations have to be reimplemented or at least libraries have to be found that provide the same functionality as MATLAB. (-) - Provides many parallel synchronisation structs.	- Static typing (+/-) - Programmer deals with memory allocation/deallocation (-) - Performance (+) - Not platform independent (-) - Find or create matrix operations library (-) - Provides many parallel synchronization structs (+) CUDA (+), but with yet unknown effort (-).

Since the existing code is already written in MATLAB and rewriting everything in either of the other languages would pose quite some risks to the project, I decided for using MATLAB in agreement with the client.

7.5.1 Object oriented programming in MATLAB

MATLAB provides object oriented features, even multiple inheritance and an event-system.

A basic difference to other object oriented languages is how object-oriented types (all types of MATLAB) behave by default.

MATLAB types such as doubles, arrays, strings, structs, cell-arrays are value types.

Reference-types (handles) are only used in a collaborative context, usually where callbacks are involved.

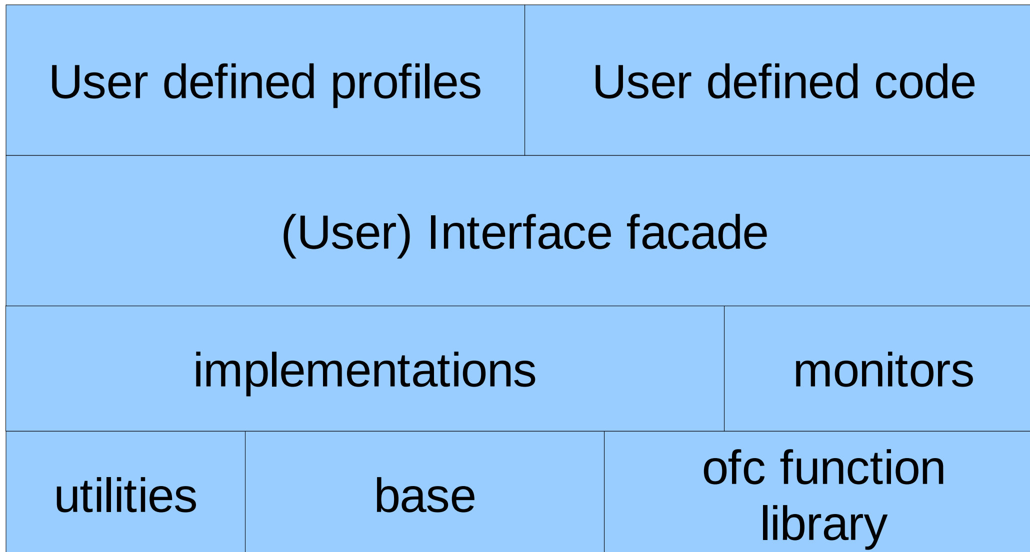
MATLAB also provides an event framework, which, by default, is not interface-based (like in Java) but function based (like in C#). Because I find the Java approach more uniform I converted these mechanisms to an interface-based approach in the implementation.

[4]

8 Implementation

This section gives an overview of the implementation. It is a starting documentation for using the algorithm.

8.1 Layer model – Design overview



1. **OFC Function Library (library)**

   This package basically consists of all numeric functions specific for optical flow calculation. Only smaller modifications were made here (See Changelog).

2. **Base**

   This package represents the object oriented model. It contains the OFCProcessor interface and the general-purpose OFCIterator class as well as data structures and superclasses for monitoring

3. **Utilities**

   Some helper functions

4. **Implementations**

   Implementation of the algorithm

5. **User interface facade**

   This package is a thin layer to allow flexible and easy access to the components of the algorithm. It is similar to an API like the unix syscall-Interface except that it passes configuration parameters through to the „kernel“ instead of function calls. It removes the need of dealing with algorithm/“kernel”-objects in the „user space“.

   It also deals with persistence of user parameters, CPU-parallelization and batch processing.

---

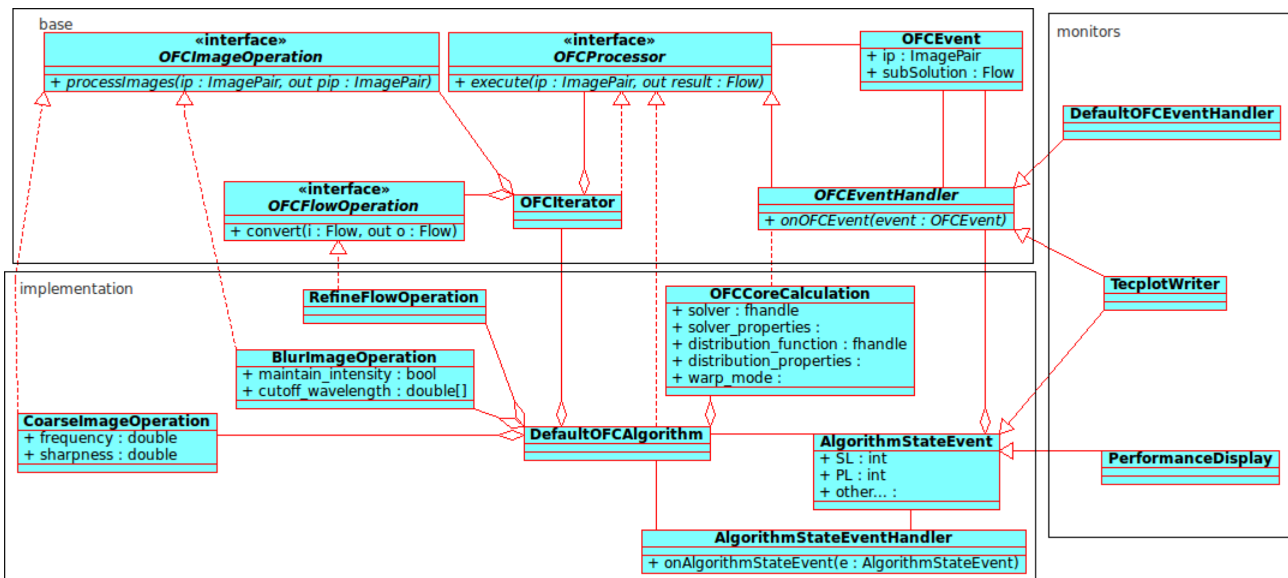
6. **User defined profiles**

   This package is subject to modification all the time. It contains functions that behave exactly like flat-file configuration files by filling a configuration struct, which can be used as maps, with simple or complex parameters. Unlike persisted profiles they are intended for a more dynamic way of use.

7. **User defined code**

   This is a placeholder package for user defined code in the project root.

8.2 Class diagram (base, implementation and monitors)



This class diagram shows the packages base, implementations and monitors in more detail.

8.2.1 Base

OFCProcessor is the interface to be implemented by all classes that solve the optical flow problem. No matter how good they perform without the help of other preprocessing classes.

OFCIterator represents the iterative process. It is constructed using start- and end level iteration identifiers plus an instance of OFCImageOperation, an additional OFCProcessor instance (the inner operation), and if needed also an instance of OFCFlowOperation.

OFCFlowOperation is the interface for operations that modify the resulting flow field of an iteration.

OFCImageOperation is the interface for operations that modify the image pair before it is passed to the inner operation.

OFCEvent is the event emitted by implementations of OFCProcessor in order to inform monitoring classes about the current state of both the input and output signal.

OFCEventHandler is the base class for all monitoring classes that don't need to know of algorithm specific parameters such as scale and pyramid level.

8.2.2 Implementation

CoarseImageOperation downsamples the image pair by a power of 2. Depending on the level of iteration.

BlurImageOperation applies a gaussian filter to the image pair and if desired normalizes the total energy of the image pair.

OFCCoreCalculation is the class that finally does the calculation of the optical flow. It is responsible for warping, alpha distribution, constructing and solving the system matrix.

DefaultOFCAlgorithm creates a complete algorithm from the implementations of the base classes. It actually creates the same object graph as in the sequence diagram in the Design section and adds a CoarseImageOperation to the outer iterator and a BlurImageOperation to the inner iterator.

It additionally introduces a more specific event system.

If requested it generates a individually configured OFCCoreCalculation instance for each iteration.

AlgorithmStateEvent is a more specific event datatype generated by DefaultOFCAlgorithm that contains all data of OFCEvent plus statistical information such as the time required for each step of OFCCoreCalculation and the pyramid and scale level.

It is emitted every time OFCCoreCalculation.execute exits.

AlgorithmStateEventHandler is the base class for all classes that wish to monitor the DefaultOFCAlgorithm.

8.2.3 Monitors

TecplotWriter writes files that can be opened using Tecplot. It extends both monitoring base classes as it needs the original images right when they enter the algorithm as well as all DefaultOFCAlgorithm specific properties such as pyramid and scale levels.

DefaultOFCEventHandler describes the algorithm state using text.

PerformanceDisplay logs performance statistics to the screen or to a file.

For more detailed descriptions and usage of the described classes, read the documentation provided with the class/function files.

8.3 User interface facade

This layer removes most of the object oriented complexity from the users scope.

However, it also introduces 2 new objects **BatchUnit** and **TestDefinition**. A BatchUnit is an item intended for execution inside a (parallel) batch loop. It is independent from other BatchUnits.

A cell array of batchunits is regarded as a **batch**.

TestDefinition is a template class to be inherited by the user. The inherited classes are used to combine a set of image pairs with a set of different algorithm configurations.

The layer introduces functions such as **parseconfig** and the persistence functions **persist** and **loadconfig** (just the algorithm instance), **loadbatchunit** (an algorithm instance plus image pairs), **loadbatch** (a series of batch units, a batch).

The function parseconfig reads a MATLAB-struct and returns a fully configured DefaultOFCAlgorithm with all requested event handlers attached.

Structs can be used very similar to maps in Java, which are often used to store flat-file configuration files.

However, in MATLAB, writing functions to construct structs is just as easy as writing into a configuration file (such as INI), but allows for much more flexibility. What is known as '#include' in C can be achieved by simply calling another function and then extending the resulting configuration struct.

Without writing any additional scanner it also allows more complex data structures to be stored within the struct such as inline function handles, arrays, matrices and other structs.

The user can also pass parameters to the function to derive configuration values from.

The folder or package 'profiles' contains some example configuration-functions and documentation.

The disadvantage of using functions as profiles is that they cannot be reconstructed from the state of an algorithm instance and therefore they are no help to build a two-way persistence (save and load) framework.

A separate solution is introduced to handle persistence.

8.3.1 parseconfig

The function parseconfig constructs or applies changes to an instance of DefaultOFCAlgorithm using the scale and pyramid level parameters of the configuration struct.

It searches all public properties of the newly created OFCCoreCalculation routine(s), the Blur- and CoarseImageOperation for matching parameters of the configuration struct.

Upon match the properties are redefined using the user-defined value instead of the default.

If desired the user can also define different properties for each iteration by using cell-arrays. Each cell then responds to a single property of the corresponding OFCCoreCalculation, selected for that iteration. Scale levels are varied along the cell row while pyramid levels are varied along the cell column if multiples are existent.

In addition it reads the 'monitors' field from the struct as a cell array and adds the contents as monitors.

Parseconfig provides a function handle which takes no arguments as a second return value. Executing this function removes all monitors added using parseconfig and enables for conflict free reexecution of the same algorithm instance.

Example profiles

```
function c = emptyprofile(index)
%EMPTYPROFILE Creates an empty profile
c = struct();
end
```

Creates a valid profile, that does not apply any changes to the default values (which are stored as default property values in the related class files)

```
function c = referenceprofile()
%REFERENCEPROFILE Reference profile with default values
c.topPyramidLevel = 4;
c.bottomPyramidLevel = 1;
c.endScaleLevel = 3;
c.pyramid_frequency = 1.5707963267949;
c.pyramid_sharpness = 0.05;
c.scale_maintain_intensity = true;
c.scale_cutoff_wavelength = [0.636619772367581 0.318309886183791];
c.scale_sharpness = [0.05 0.05];
c.core_solver_residual = 0.0001;
c.core_solver_max_iterations = 2000;
c.core_alpha = 0.3;
c.core_alpha_variation_scale = [0.125 0.1];
c.core_gradient_factor = 0;
c.core_velocity_factor = 0;
c.core_laplace_velocity_factor = 0;
c.core_alpha_distribution = 1;
c.core_gradient_factor_x = 0;
c.core_gradient_factor_y = 0;
c.core_velocity_factor_x = 0;
c.core_velocity_factor_y = 0;
c.core_laplace_velocity_factor_x = 0;
c.core_laplace_velocity_factor_y = 0;
c.core_alpha_variation_scale_x = [0.125 0.1];
c.core_alpha_variation_scale_y = [0.125 0.1];
c.core_alpha_zero = 0.4;
c.core_omega1 = 0.04;
c.core_omega2 = 0.04;
c.core_lambda = 0.2;
c.core_background = 0;
c.core_delta_p = 1;
c.core_delta_t = 1;
c.core_warp_mode = 'warp_both_frames';
c.core_solver = @bicgstab;
end
```

This profile contains all values, but they are all set to default, so parseconfig(referenceprofile) returns the same as parseconfig(emptyprofile)

```
function c = variationexample()
c.core_alpha = {0.3 0.4 0.5; 0.31 0.41 0.51; 0.32 0.42 0.52; 0.33 0.43 0.53};
c.core_alpha_distribution = 4;
c.core_solver = {@pcg;@bicgstab;@bicgstab;@bicgstab};
c.varyscale = true;
c.varypyramid = true;
end
```

In this example it is assumed, that the default iterations count for pyramid levels is 4 and the count for scale levels is 3.

This profile creates a configuration where the alpha property of OFCCoreCalculation is varied along both pyramid and scale levels during execution.

The solver which is used to solve the equation is changed during execution. pcg will be used for the top pyramid level and bicgstab for the remaining iterations.

Variation over scale levels is achieved by defining multiple colums while variation over pyramid levels is achieved by definining multiple rows.

Because parameters are varied varyscale and varypyramid are set to true.

```
function c = vsj_statistics(pc)
deltat_list=[1 3 1/3 1 1 1 1 1]*.033;
deltat = deltat_list(pc.index);


vsj_file = [pc.input_path '/Standard_VSJ_velocity_data.txt'];
c.monitors = {TecplotWriter VSJErrorDisplay(vsj_file,deltat)};

c.monitors{1}.writeDir = pc.output_path;
c.monitors{1}.analytic_exists = true;
c.monitors{1}.analytic = readvsjanalytic(vsj_file,deltat);
c.monitors{2}.writeFile = [pc.output_path '/VSJ_ERROR.log'];
end
```

This function only defines monitors. It is used to test the algorithm using the Standard VSJ image pairs. In order for the monitors to test the results, they are in need of the original vector field that was used to generate the image pairs and the time interval (deltat) between the images they are going to monitor.

8.3.2 Persistence

The function persist writes either a cell array of BatchUnits, a BatchUnit or DefaultOFCAlgorithm to a file or filedescriptor, overwriting existing files. By defining a namespace multiple instances can be written to one filedescriptor without causing conflicting entries. Monitors attached to the instance are ignored.

Example generated by using persist on a cell array of BatchUnits:

```
#################################
## START OF BATCH UNIT BU0001 ##
##################################
bu0001.image1 = '/home/testi/Desktop/vsj_read/piv01_1.bmp'
bu0001.image2 = '/home/testi/Desktop/vsj_read/piv01_2.bmp'
bu0001.topPyramidLevel = 4
bu0001.bottomPyramidLevel = 1
bu0001.endScaleLevel = 3
bu0001.pyramidGenerator.frequency = 1.5707963267949
bu0001.pyramidGenerator.sharpness = 0.05
bu0001.scaleGenerator.maintain_intensity = true
bu0001.scaleGenerator.cutoff_wavelength = [0.636619772367581 0.318309886183791]
bu0001.scaleGenerator.sharpness = [0.05 0.05]
bu0001.coreRoutine.solver_residual = 0.0001
bu0001.coreRoutine.solver_max_iterations = 2000
bu0001.coreRoutine.alpha = 0.3
bu0001.coreRoutine.alpha_variation_scale = [0.125 0.1]
bu0001.coreRoutine.gradient_factor = 0
bu0001.coreRoutine.velocity_factor = 0
bu0001.coreRoutine.laplace_velocity_factor = 0
bu0001.coreRoutine.alpha_distribution = 1
bu0001.coreRoutine.gradient_factor_x = 0
bu0001.coreRoutine.gradient_factor_y = 0
bu0001.coreRoutine.velocity_factor_x = 0
bu0001.coreRoutine.velocity_factor_y = 0
bu0001.coreRoutine.laplace_velocity_factor_x = 0
bu0001.coreRoutine.laplace_velocity_factor_y = 0
bu0001.coreRoutine.alpha_variation_scale_x = [0.125 0.1]
bu0001.coreRoutine.alpha_variation_scale_y = [0.125 0.1]
bu0001.coreRoutine.alpha_zero = 0.4
bu0001.coreRoutine.omega1 = 0.04
bu0001.coreRoutine.omega2 = 0.04
bu0001.coreRoutine.lambda = 0.2
bu0001.coreRoutine.background = 0
bu0001.coreRoutine.delta_p = 1
bu0001.coreRoutine.delta_t = 1
bu0001.coreRoutine.warp_mode = 'warp_both_frames'
bu0001.coreRoutine.solver = @bicgstab
##################################
```

The function recursively calls itself and makes use of the namespace parameter to allow multiple batch units to be written. **bu0001** is a top level namespace used.

To load the cell array of batchunits defined above one would call:

batchCell = loadbatch(filename)

if a single batch should be read, then one would call

bu = loadbatchunit(filename, 'bu0001')

for the first batchunit

8.3.3 Test definitions

The class TestDefinition is an abstract class that only implements a single method run().

It is used by inheriting it, creating an instance of the inherited class and execute run() on that instance.

Its purpose is to combine a set of image-pairs and a set of configurations with certain monitors applied.

Between the set of image-pairs and the set of configurations, the cartesian product is built, so that each image pair is passed to each algorithm configuration.

The method run() finally generates the batch and executes it. If desired, the user can disable parallel execution by setting the parallel-property to false.

```
    methods (Abstract)
        %Returns a profile struct for the e'th element of B
        profile = getProfile(obj,e)



        %Returns the image pair descriptors (usually paths) for the e'th
        %element of A
        [path1, path2] = getPair(obj,e)



        %Returns the monitoring profile for the eA'th element of A and the
        %eB'th element of B
        mprofile = getMonitor(obj,eA,eB)


        %Returns the name of the directory where monitoring results are to
        %be stored. It may be empty if no files need to be stored.
        mdir = getMonitorDir(obj,eA,eB)

        %Returns the root dir where all batch items are stored. It may be
        %empty.
        mdir = getMonitorRoot(obj)

    end
```

Snippet of the abstract class TestDefinition. A declares the set of image-pairs while B declares the set of profiles.

The user has to define the properties pairCount = |A| and profileCount = |B| and implement the abstract methods.

Function	What the function-body needs to do
getProfile	Return a struct with the same structure as returned by the functions in the folder profiles, or return a struct with the same structure as returned by the correctly namespaced loadtree() function. Depending on the parameter e, the function should return a different configuration. The parameter e ranges from 1 to profileCount.
getPair	Return the path to 2 images. Different paths depending on the parameter e, which ranges from 1 to pairCount. If IM7-files are read, the returned path may has ':1' or ':2' appended to address a certain image inside an IM7-file.
getMonitor	Return a struct with only the field monitors defined. The field monitors must be a cell array containing instances of OFCEventHandler or AlgorithmStateEventHandler. The user may let the configuration vary depending on the parameter eA, corresponding to parameter e in getPair, and/or parameter eB, corresponding to to the parameter e in getProfile. The user should use the function getMonitorDir to define the paths were the monitors store their results.
getMonitorDir	Return the path (not relative to getMonitorRoot()) to where monitoring results for an algorithm execution are stored. The function should return different results depending on the parameter eA, which corresponds to the parameter e in getPair, and the parameter eB, which corresponds to the parameter e in getProfile. It is up to the user to define the path structure. The paths are created on execution, no matter how deep the folder structure is going to be. The result may be empty for the case that monitors do not write to files.
getMonitorRoot	Returns the directory to where overall statistics of the TestDefinition should be stored. It may return an empty value.

VSJTestDefinition

An example-implementation exists, that varies the way the **smoothness condition** is built

VSJTestDefinition pairCount = 8 profileCount = 4 parallel = true	VSJ Pair 01	VSJ Pair 01	VSJ Pair 02	VSJ Pair 03	VSJ Pair 04	VSJ Pair 05	VSJ Pair 06	VSJ Pair 07
core_alpha_distribution = 1	BU 01	BU 02	BU 03	BU 04	BU 05	BU 06	BU 07	BU 08
core_alpha_distribution = 2	BU 09	BU 10	BU 11	BU 12	BU 13	BU 14	BU 15	BU 16
core_alpha_distribution = 3	BU 17	BU 18	BU 19	BU 20	BU 21	BU 22	BU 23	BU 24
core_alpha_distribution = 4	BU 25	BU 26	BU 27	BU 28	BU 29	BU 30	BU 31	BU 32

This test definition stores its results in '<project root>/data/vsj_results_extended_??'.

Each row in this table receives an own subdirectory and each column-cell of a row a subdirectory in the corresponding subdirectory.

When the method run() is executed, a cell array of batch units is created with a length of 32 entries.

MATLABs parfor-loop then does its best to distribute these 32 batch units to the worker processes.

Should there be changes to the code, which do not affect the monitoring configuration and the amount of profiles/pairs, then an old test can be rerun by executing:

VSJTestDefinition().run('<project root>/data/vsj_results_extended_??/batch.profile');

8.4 Parallelization

In order to speed up computation, two distinct parallelization approaches were evaluated, CPU and GPU-Parallelization. CPU-parallelization is achieved by using multiple CPUs or multiple cores of a CPU concurrently. GPU-Parallelization is achieved by delegating vectorized tasks to the GPU. The GPU executes in parallel by nature.

8.5 CPU-Parallelization

MATLAB provides the parfor construct to replace for-loops were each loop is independent from previous loops. This approach was successfully used to execute batch jobs in parallel. MATLAB does not use Threading in order to allow execution on computing clusters, thus the communication overhead was observed to be slowing down the algorithm for fine-granular parallel calculations (See Testing).

For batch processing, however, this approach became useful, because only very sparse data is required to be communicated between batch management and batch unit execution.

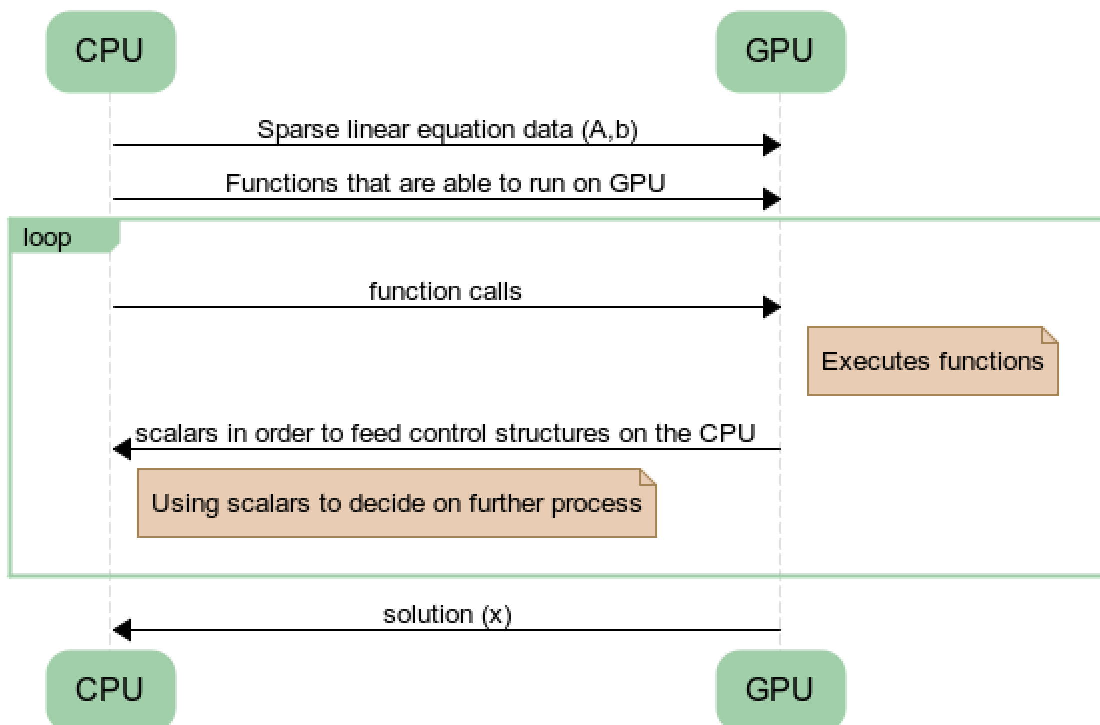
To use multiple CPU cores concurrently '**matlabpool open**' has to be used on the MATLAB-commandline.

The amount of workers available can be looked up using '**matlabpool size**'

[5]

8.6 GPU-Parallelization

Since parfor seemed unusable for fine-granular calculations other options were investigated. Now, the implementation provides the possibility to move the linear system to GPU memory. The unmodifed bicgstab function then operates on these „remote“ arrays while the function itself resides on the CPU. It is not possible to store the function on the GPU, because the available CUDA-MATLAB APIs do not allow functions with control structures to reside in GPU memory. However, since many builtin-functions that bicgstab consists of can run on the GPU the communication overhead is minimal (See Testing for Performance comparison). Also the effort for this implementation was minimal as it basically consists of injecting typecasts.



CPU, GPU interaction

Simplified model of how the CPU and GPU interact when solving a linear equation using bicgstab.

There were 2 options to integrate CUDA into the project. One is provided by Mathworks and is integrated since MATLAB2010b, the other approach is provided by a third party (AccelerEyes) called Jacket. Jacket is a commercial solution with additional licence fees.

The advantage of using Jacket is that it supports sparse linear algebra, while the integrated solution only supports creating dense arrays on the GPU. Sparse linear algebra is required by the project, because the problem size would grow quadratically with dense arrays.

[6][7][8]

Since there are additional license fees after the trial expires CUDA-Support is disabled by default and only used to demonstrate the potential power of. It can be activated for the linear equation solver by setting the profile configuration flag as follows:

c.core_solver = @gpu_bicgstab

Depending on the GPU speed, PCI-E Bus transfer rate and CPU speed you may specify the flag as follows (assuming 4 Pyramid Levels):

c.core_solver = {@bicgstab;@bicgstab;@gpu_bicgstab;@gpu_bicgstab};

c.vary_pyramid = true;

This is probably the optimal configuration for a 256x256 image pair, because on an average system the gpu computing speed can't compensate the bus transfer costs for the first iterations.

The function **gpu_bicgstab** is a wrapper around bicgstab that converts double-arrays lying in the cpu-workspace into gdouble-arrays on the GPU before execution.

Eventually more than just bicgstab can be optimized using GPU-arrays. A flag/property **gpu_optimization** has been introduced to OFCCoreCalculation. As of now there are still some problems to using this flag, because some functions use the function **double**, which transfers gdouble-arrays back to CPU-Memory.

8.7 Walkthrough & Use Cases

This section will walk you briefly through the whole solution, demonstrating some use cases. The idea is to provide an understanding through all architectural layers.

The Walkthrough chapters are independent from each other. You may start where you wish.

At first the environment needs to be set up correctly:

1. Change the directory to '<project root>/src'

   For example: cd /home/user/matlab_projects/ofc/src
2. Open the file init.m and let it point to your installation of Jacket (if desired).
3. Run init

1. Setting up an instance of the algorithm manually

   Starting point for understanding the structure of the algorithm and learning how to apply structural changes to the algorithm
2. **Using the 'shipped' algorithm class**

   Explains how the standard algorithm for optical flow calculations can be configured and executed directly, without using the interface-functions.
3. **Profiles**

   Explains how profile-structs can be used to configure the algorithm.
4. **Persistence**

   Explains how an algorithm instance can be saved to and loaded from the filesystem. This also includes batches.
5. **TestDefinitions**

   Explains batch processing by combining a set of image pairs and a set of profiles.

8.7.1 Setting up an instance of the algorithm manually

```
% First we need to instantiate the core routine. This is the component,
% which does the actual calculation.
c = OFCCoreCalculation()

% If you wish you can edit the new instance, for example you can set the
% solver to gpu_bicgstab instead of the default bicgstab
% (only works with Jacket installed)
c.solver = @gpu_bicgstab

% You could now already run that instance using an image pair. the last
% argument is the initial guess, but we don't have one, so we just define it
% as zero.
i1 = imread('../data/vsj/piv01_1.bmp');
i2 = imread('../data/vsj/piv01_2.bmp');
result = c.execute(i1,i2,0);

%You can monitor the progress of c, by adding a listener to it:
e = ExampleOFCEventHandler
e.listenTo(c)

% The result is a column-vector, which needs to be converted in order to
% display it.
[u,v] = sol2uv(result,256,256);

% U declares horizontal movement, V declares vertical movement

% Display horizontal movement, you need to multiply the result, because the
% values are unnaturally small
image(u*1e16)

% As you can see nothing that looks like a smooth movement is displayed.
% This is because the core routine needs a well prepared input.
% That's why we now construct the iterators.

% First we need to instantiate the "filter" we'd like to apply
% We instantiate BlurImageOperation to be prepared for up to 3
```

```
% scalelevel iterations.
s = BlurImageOperation(3)

% For demonstration you can apply the filter as follows to the image pair
% This will create the filtered image pair for the second iteration
% (scale level 2).
[pi1, pi2] = s.processImages(i1,i2,2);

% In "production" BlurImageOperation is not used directly, instead we create
% an iterator that makes use of it:

sli = OFCIterator(1,3,s,c)

% OFCIterator provides the same execute method as OFCCoreCalculation
% sli will loop from 1 to 3, applying BlurImageOperation, executing the
% core calculation, gather its flow results and reuse these for the next
% iteration.

% Monitoring (if desired)
e.listenTo(sli)

% Do the calculation
result = c.execute(i1,i2,0);

% You should see that the results are
% smoother and not near zero
[u,v] = sol2uv(result,256,256);

image(u*10)
image(u*10)

max(max(u))

% max(max(u)) should result in 5.5961
% We are not sure that the maximum horizontal movement in that image pair
% is only 5-6 pixels, thus we introduce an additional iterator:

p = CoarseImageOperation()

% The filter downsamples an image pair of resolution WxHx2 to
% (W / 2^(i-1)) x (H / 2^(i-1)) x 2, where i is the iteration identifier
% (pyramid level).
% 1 -> 256x256x2, 2 -> 128x128x2, 3 -> 64x64x2, 4 -> 32x32x2

% The new iterator is going to start at level 4 (using the smallest image
% pair), the resulting flow field will have the same resolution.
% Because the next iteration will run at a higher resolution, but depends
% on the result of the previous resolution, we need to upsample the result
% to fit the new iteration. We introduce RefineFlowOperation.


r = RefineFlowOperation([256 256],4)

% Its awkward that this operation requires the image size and the top
% pyramid iteration level, but currently can't be helped.



% Construct the iterator, additionally with the RefineFlowOperation:
pli = OFCIterator(4,1,p,sli,r)

%  This iterator behaves just like the scale level iterator, but instead of
%  passing the filtered images to the core calculation it passes them to the
%  scale level iterator.

% Monitoring
e.listenTo(pli)

% Calculation:
```

```
result = pli.execute(i1,i2,0);

[u,v] = sol2uv(result,256,256);

max(max(u)) % == 14.1600

% Now the algorithm has been constructed completely. The class
% DefaultOFCAlgorithm is basically the same as what we constructed here.
```

8.7.2 Using the 'shipped' algorithm class

```
% Instantiate the algorithm with a default configuration

a = DefaultOFCAlgorithm

% Instantiate a monitor that displays time statistics.
% This time we don't use a monitor that observes the call stack,
% but a performance monitor that takes a "snapshot" after each execution of
% OFCCoreCalculation
pm = PerformanceDisplay

% Add the monitor
pm.listenToState(a)

% Load the images:
i1 = imread('../data/vsj/piv01_1.bmp');
i2 = imread('../data/vsj/piv01_2.bmp');

a.execute(i1,i2,0);

% By default an algorithm with 4 pyramid levels and 3 scale levels is
% instantiated. The constructor allows for individual configuration:

a = DefaultOFCAlgorithm(5,2,4,true,true)
pm = PerformanceDisplay
pm.listenToState(a)

% Now the algorithm iterates from pyramid level 5 (16x16x2) to pyramid
% level 2 (128x128x2) and scale level 1 to 4
% The 2 booleans added to the constructor, cause to create an
% OFCCoreCalculation instance for every single iteration.
a.execute(i1,i2,0)

% You may alter the properties of the core calculations by accessing the
% contents of the cell-array property coreRoutine of a
% Variation along rows means variation along scale levels
% Variation along columns means variation along pyramid levels
% top and left cells are executed first
```

8.7.3 Using profiles

```
% A profile is nothing but a struct. It may be empty.
c = struct()

%This will generate the default algorithm
a = parseconfig(c)

%Do some alterations
c.core_alpha = 0.4
a = parseconfig(c)

a.coreRoutine{1}.alpha

% The field core_alpha was translated into the core routines property alpha

% We now alter alpha along scale levels, by default there are 3 scale
% levels -> 3 columns
c.core_alpha = {0.3 0.4 0.5}
% Tell parseconfig to allocate multiple core instances
c.varyscale = true;

a = parseconfig(c)

a.coreRoutine{:}

% You can also alter along pyramid levels
c = struct()
c.core_alpha = {0.3; 0.4; 0.5}
c.varypyramid = true;

% Or both (by default there are 4 pyramid levels -> 4 rows)

c.core_alpha = {0.3 0.4 0.5; 0.31 0.41 0.51;0.32 0.42 0.52; 0.33 0.43 0.53}
c.varypyramid = true
c.varypyramid = true

% Alter the amount of pyramid levels
c = struct()
c.topPyramidLevel = 5

a = parseconfig(c)

% You may alter an existing algorithm
c = struct()
c.core_solver = @gpu_bicgstab
c.monitors{1} = PerformanceDisplay
[~,cleanup] = parseconfig(c,a)
i1 = imread('../data/vsj/piv01_1.bmp');
i2 = imread('../data/vsj/piv01_2.bmp');

a.execute(i1,i2,0)

%Detach the monitor from the algorithm
cleanup()
```

8.7.4 Persistence

```
% Create an algorithm instance
a = DefaultOFCAlgorithm()

%Display configuration
persist(a,1)

%Write to file
persist(a,'myprofile.profile')

%Load from file
a = loadconfig('myprofile.profile')

%Persistence for batch units
davis =
'../data/davis/PIV_PANDA_Cooler_Series_St4_x_New/N011_PosG/S06_PosG_N1024_f05_Lm10_PIII/B00001.im7
';

bu = BatchUnit(struct,[davis ':1'],[davis ':2'])

%Overwrite previous configuration
persist(bu,'myprofile.profile');

bu = loadbatchunit('myprofile.profile')

%Apply a non-persisted profile (used to add monitors usually)
c.monitors{1} = PerformanceDisplay
bu.applyProfile(c)

%Execute (IM7-format only works on Windows systems currently)
result = bu.execute();

%Store batchunit in a cell array. In that case persist will look at it as a
%series of batch units. You may extend the cell array using additional
%batch units.

bus = {bu}

persist(bus,'myprofile.profile')

bus = loadbatch('myprofile.profile')
```

8.7.5 Working with TestDefinitions

```
% Instantiate the VSJ Test Definition

v = VSJTestDefinition()

% Run the getter methods of v to see which elements this test definition is
% going to use to built the cartestian product on and run the batch test
% on.

[p1,p2] = v.getPair(1)
[p1,p2] = v.getPair(8)

v.getProfile(1)
v.getProfile(4)

% The profiles generated here need either be compatible with parseconfig() or
% loadtree().

v.getMonitor(1,1)
v.getMonitor(4,2)
v.getMonitor(6,3) %Monitor configuration for 6th image pair, 3rd profile
v.getMonitor(8,4)

v.getMonitorRoot()

%Decide whether to run parallel or not
v.parallel = true

matlabpool open %Needed for parallel execution


% Run the test
v.run()

% In case you altered the methods getPair or getProfile, but wish to
% execute an older test profile:

v2 = VSJTestDefinition()
v2.run('../data/vsj_results_extended_01/batch.profile')


% Write your own test definition, for example by reading the contracts of TestDefinition.m and
% copying and altering VSJTestDefinition.m
```

8.8 Changelog

This section lists all changes that were made to the existing solution.

Changes were made that make the user-interfacing functions unusable (TopLevelCheck), because global variables were removed entirely.

The paths given here are relative to '<project root>/src/library'

Add_On/distribute_alpha.m	The function requires an additional parameter conf, a struct containing fields that previously were globals. The function no longer uses globals. This change is incompatible with TopLevelCheck*, OPTICAL_FLOW_SCRIPT and OPTICAL_FLOW_CALCULATION3
Add_On/distribute_smoothness.m	Same as distribute_alpha.m
Add_On/distribute_smoothnes_xy.m	Same as distribute_alpha.m Call to write_plot_files and function-argument Current_pair removed
Add_On/writeTecPlot_file.m	Vector dY is no longer multiplied by -1 for compatibility with absolute values. function user is responsible for that step now, if necessary.
pivmat	Updated pivmat to version 2.01 Moved to '<project root>/lib'
readIMX	Updated readimx to version 1.5R1_2009 Moved to '<project root>/lib'
W_Prog/BUILD_PYRAMID_LEVEL.m	This file is new. Replacement for BUILDING_PYRAMID.m. The function takes an image array as an argument instead of a file.
W_Prog/warp_image_V04.m	Optionally takes a boolean 4th argument (PARALLEL) in order to decide for parallel warping.
W_Prog/scaling_gray_values.m	Accepts a struct instead of globals

8.9 Implemented requirements

This section covers how far the clients requirements could be implemented.

ID	Priority	Name	Progress	Progress description
C01	High	Extendability and Modifiability	100%	All refactoring issues could be adressed. Some global variables may still exist, but it is not required for the user to modify them and they don't conflict with parallelization.
C02	Medium	Performance	100%	As of now parallelization has been implemented for batch processing and bicgstab has been improved to run on the GPU.
C03	Medium	Usability	90%	With parseconfig, persistence and profile functions a flexible way of accessing the whole algorithm has been introduced, but usability is not tested. All classes and functions are documented.
C04	High	Accuracy	95%	The framework for testing accuracy is established and tests were executed. The results seem satisfying, but are not equal to the previous solution.

ID	Priority	Name	Progress	Progress description
C05	High	Tecplot Monitor	100%	A Tecplot writer has been implemented and tested.
C06	High	Performance Monitor	80%	A Performance Monitor has been implemented and tested, but does not print results based on the call hierarchy, but detailed statistics about relevant modules.
C07	High	Batch processing	100%	With TestDefinition and BatchUnit a complete batch processing framework is implemented.
C08	Low	Cross correlation	0%	Not implemented.
C09	High	Single Run Profiles	100%	Also covered by parseconfig. Documented examples exist.
C10	High	Batch Profiles	90%	Perstistence layer implemented for writing and reading. The persistence layer is currently missing support for monitor-configuration.
C11	Medium	Read image files	80%	Without using pivmat, but using readimx directly reading davis files has been implemented. But support is very basic and not elaborated.
C12	Low	GUI	0%	Not implemented.

8.10 Remaining issues

Some properties of OFCCoreCalculation are only used if the property 'alpha_distribution' is set accordingly. A better grouping of these properties should be implemented.

The persistence layer (persist and load*) currently does not support to save monitor configurations to files. The monitoring information currently has to remain in code, for example inside TestDefinitions.

Monitoring for TestDefinitions currently also lacks summarizing monitoring results. Currently each batch unit execution receives its own subdirectory inside the filesystem to store its result in an isolated manner. A summarizing extension to the monitoring system should allow for more detailed overall statistics for a TestDefinition or its underlying subsets (See TestDefinition).

The class OFCPadding, responsible for modifiying the image pair to fit the iterative downsampling process currently also modifies the flow field to match the original images, but this is not yet tested.

The support for the DaVis/LaVision is restricted to IM7-files and currently does not support SET-files.

9 Testing

As mentioned there are 2 important items to be tested, Performance and Accuracy.

9.1 Performance

Because we introduced some parallelization options to the algorithm these have to be tested. This section also covers items that do not meet the requirements and thus are disabled in the current solution.

The items to test are:

- Parallel warping
- GPU-optimized bicgstab (solver)
- Parallel batch processing

9.1.1 Parallel warping

Testing environment

OS: Ubuntu 10.10 i386

MATLAB-Version: MATLAB 2010b

CPU: AMD Athlon(tm) 64 X2 Dual Core Processor 5200+

Test configuration

matlab poolsize: 2

Function: `ParallelVsNonParallelWarpingTest`

Note that the tests were not executed using TestDefinitons, because the class was not implemented at that time.

This function creates a batch of 10 tests. All tests run on th VSJ Standard Image pair 1 (see Accuracy tests). The first 5 tests use sequential warping of the images while the last 5 tests use parallel warping.

Only runtime is observed.

The class PerformanceDisplay is used as a monitor.

Average warp time per iteration at different Pyramid Levels in seconds

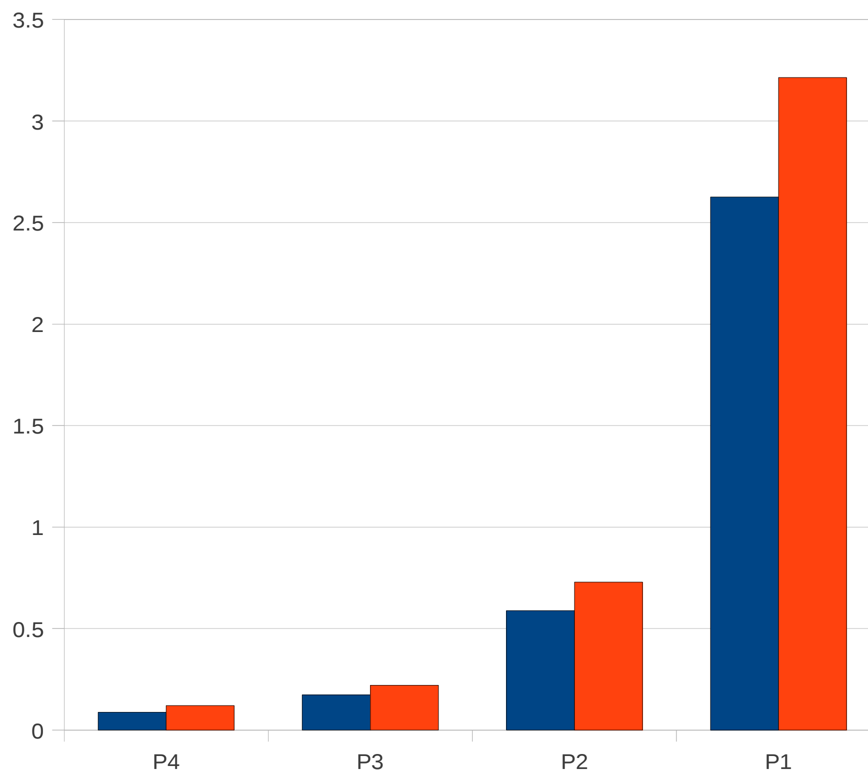
Problem sizes in pixels

P4: 32x32x2

P3: 64x64x2

P2: 128x128x2

P1: 256x256x2



Blue: Non-parallel warping

Red: Parallel warping

Warping is slower in parallel. It seems that the non-threaded nature of MATLAB, which requires interprocess communication and serialisation of objects in order to allow distributed calculations, slows synchronisation down and makes parfor not easily usable for fine-granular parallel processing. Thus we decide to use parfor for coarse-grained parallel jobs such as batch processing only.

These results led me to the decision to move the fine-grained parallel strategy for this project to SIMD processing (CUDA, OpenCL).

For the full report see '<project root>/data/performance_results_parallel_vs_non-parallel_warping'

9.1.2 bicgstab on the GPU

Testing environment

OS: Ubuntu 10.10 i386

MATLAB-Version: MATLAB 2010b

CPU: AMD Athlon(tm) 64 X2 Dual Core Processor 5200+

GPU: GeForce GTS 450, 1760 MHz, 1024 MB VRAM, Compute 2.1 (single,double)

Jacket v1.7

CUDA 3.2

Function: `CPUVSGPUTest`

This function creates a batch of 10 tests. All tests run on th VSJ Standard Image pair 1 (see Accuracy tests). The first 5 tests use bicgstab as solver while the last 5 tests use gpu_bicgstab as solver which internally creates GPU arrays.

Only runtime is observed.

The class PerformanceDisplay is used as a monitor, additionally for this test the function gpu_bicgstab has been injected for that test with code to write its runtime down, because the framework (PerformanceDisplay) doesn't support recursing into each function for runtime observation.

For the full report see '<project root>/data/performance_results_gputest'

Average solver time per iteration at different Pyramid Levels in seconds

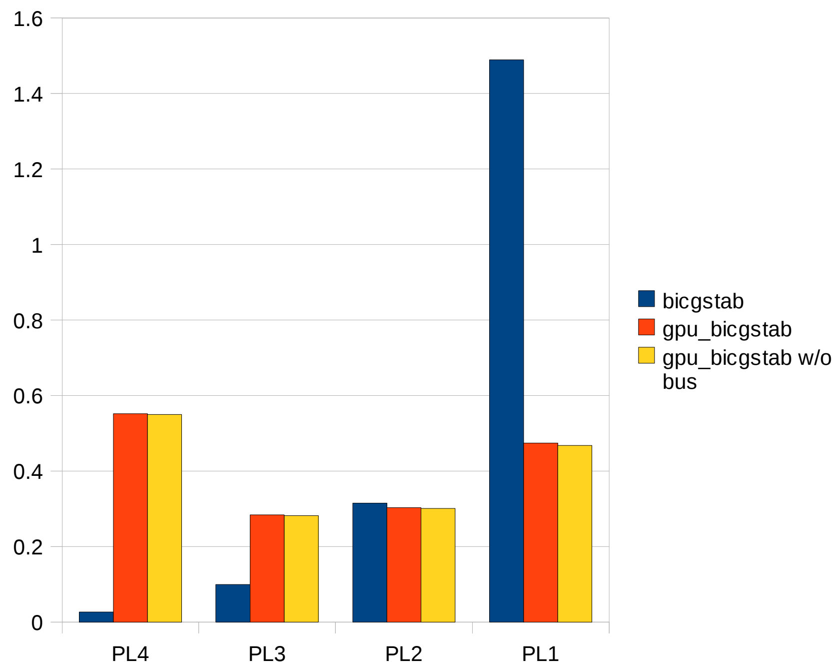
Schematic problem size: (2 x N) x (2 x N)

P4: N = 32x32

P3: N = 64x64

P2: N = 128x128

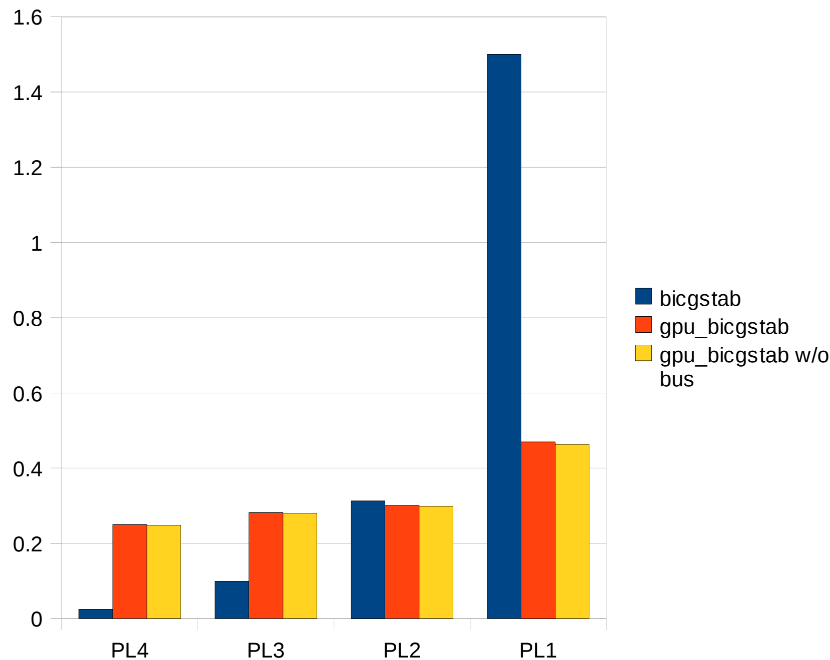
P1: N = 256x256



'gpu_bicgstab w/o bus' tries to display the calculation time excluding bus transfer time. It was discovered that Jacket uses lazy evaluation[6][9][10], so that the actual transfers may not occur at the expected line of code, rendering the results of 'gpu_bicgstab w/o bus' useless. To get corrected results the jacket function 'gsync'[11] must be called prior to recording the first timestamp.

The very first time gpu_bicgstab was executed it required about 17 times more time than its comparable iteration in the second test running gpu_bicgstab. Thus a second diagram compares the results using the median over all tests.

Median solver time per iteration at different Pyramid Levels in seconds



To better see how gpu_bicgstab performs at even higher image resolutions another test was executed. It uses an image pair of size 1024x1024x2 and uses up to 6 pyramid levels.

The image pair is stored in '<project root>/data/1024'.

Average solver time per iteration at different Pyramid Levels in seconds (1024x1024)

Schematic problem size: (2 x N) x (2 x N)

P6: N = 32x32

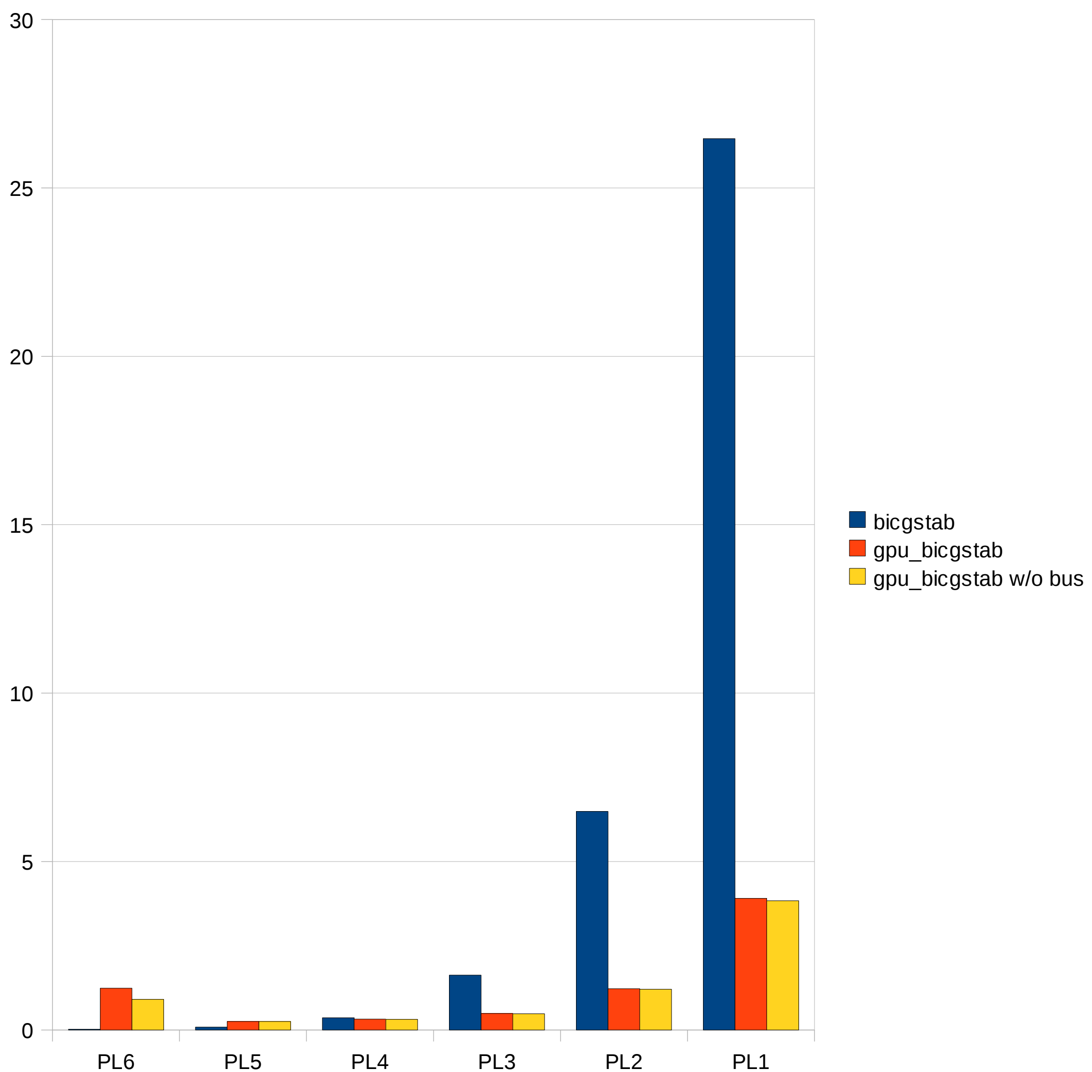
P5: N = 64x64

P4: N = 128x128

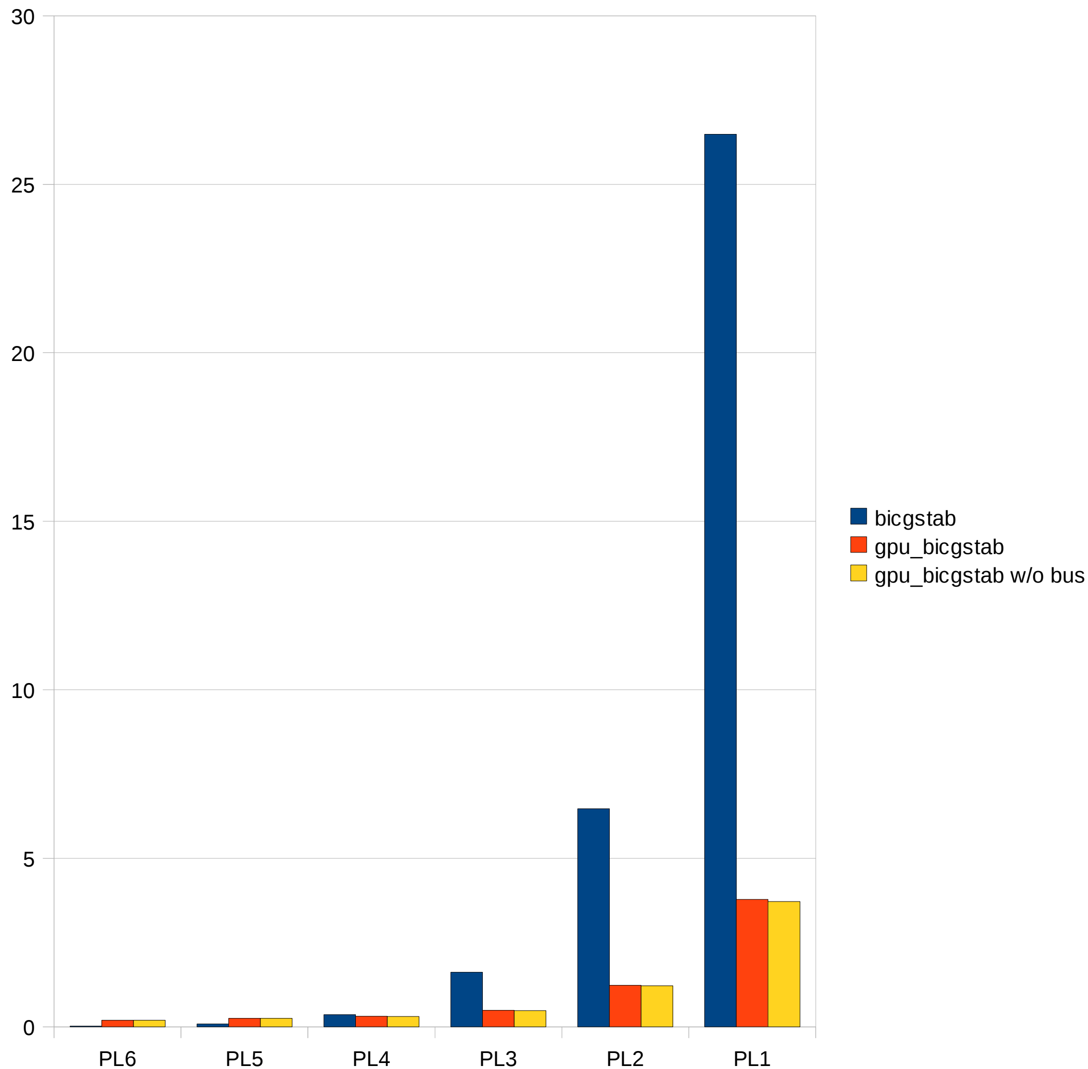
P3: N = 256x256

P2: N = 512x512

P1: N = 1024x1024



Median solver time per iteration at different Pyramid Levels in seconds (1024x1024)



Conclusion

For small problem sizes the GPU should not be used. The GPU also seems to require some warm-up time, so a single execution of the algorithm may only be faster with the GPU if the image pairs are large enough. In all other cases it is worth using the GPU (batch processing, large image pairs).

On the computer of the client, these results could not be achieved. It is assumed, that his GPU is not powerful enough or that his CPU is more efficient than the CPU on this system.

9.1.3 Parallel batch-processing

To test how well batch-processing performs in comparision to non-parallel batch-processing a system with two processors and each of it with 4 cores was used.

The chosen test set are the 8 Standard VSJ image pairs. No other monitors than PerformanceDisplay are applied. A relatively small test set is used to show how well parallel processing deals with initialization overhead.

Testing environment

OS: Windows 7 64bit

MATLAB-Version: MATLAB 2010a

CPU: Intel Xeon E5620, 2x4 Cores, 16 Threading Units

Test configuration

matlab poolsize: 8

Sequential processing: 191.5 seconds

Parallel processing: 34.7 seconds

Parallel processing is 5.5 times faster than sequential processing. The best possible result is 8.

This is a good result when we take into account, that interprocess communication is involved and that the test set is small and at best can be shared to a maximum of 8 cores.

9.2 Accuracy

Accuracy is tested using the 8 standard image pairs from the Visualization Society of Japan. To these image pairs an analytic solution exists which can be compared to the result of the calculation.

The analytic solution is a vector field: $a(x,y) \rightarrow (u,v), x,y \in \mathbb{N} x,y \leq 32$

The vector (u,v) depicts the velocity in pixels of a 256x256 image per second.

The resulting vector field has a higher resolution than the analytic vector field, thus the resolution is adjusted by selecting only 32*32 vectors at comparable positions. A vector field results

$r(x,y) \rightarrow (u,v), x,y \in \mathbb{N} x,y \leq 32$

The vector (u,v) depicts the distance in pixels of a 256x256 image instead of the velocity.

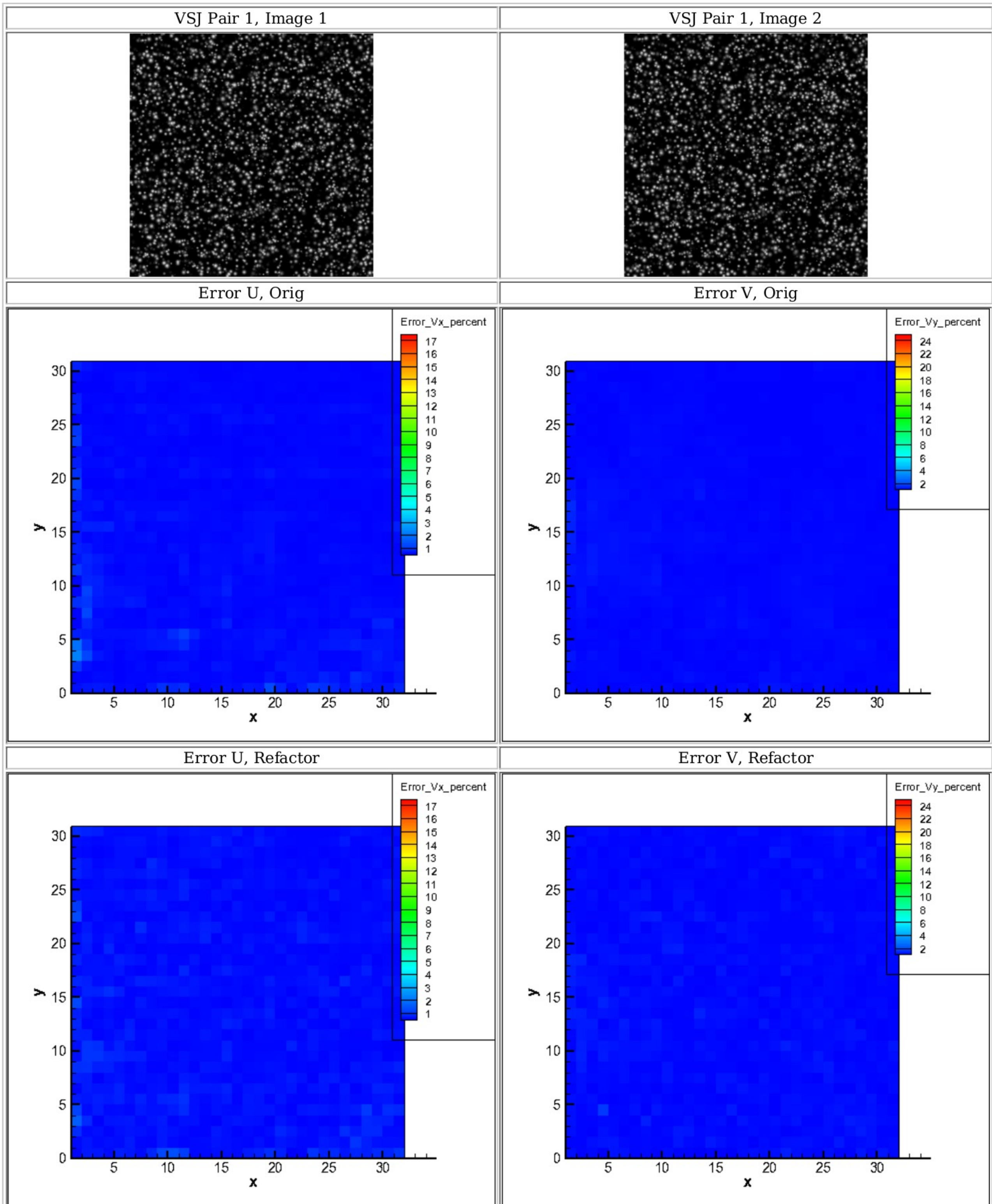
The error function is defined as follows:

$$e(x,y)=|a(x,y)\cdot dt-r(x,y)|$$

where dt is a discrete time interval between the images.

The time interval varies as follows:

VSJ pair	**dt**
01	0.033s
02	0.099s
03	0.011s
04	0.033s
05	0.033s
06	0.033s
07	0.033s
08	0.033s



Orig = Unrefactored solution, default configuration

Refactor = Refactored solution

The full report is contained as a table in 2 image files in the directory '<project root>/data/vsj_report'

The table displays the error in U and V direction for both the original algorithm as well as the refactored algorithm.

Conclusion

It seems that the error of the refactored solution has a similar structure, but the error seems to be slightly higher in the refactored solution, which shows up as higher noise.

It was detected that an image preprocessing function **scaling_gray_values** was still using global parameters, that were not defined. The function normalizes the intensity values of the image pair into a range between 0 and 1.

The function was used by CoarseImageOperation, but is now invoked using a separate OFCProcessor (**OFCPreprocessor**), that configures the function correctly.

A second test has not been executed yet.

10 Reflection

In the beginning of the project, the refactoring seemed quite easy and a first prototype was implemented after a few weeks. I was afraid that the amount of remaining tasks would not be enough. Realizing that the client does not only need the algorithm to work correctly, but also needs the monitoring code to be transferred to the refactored solution as complete as possible, was a relief, but also required deeper and difficult investigations, because I did not understand the purpose of every task. I started working on the monitors, but I had the impression that I was copying code from functions to classes without understanding it, which made the task a little boring. Later, however, the client asked me to document accuracy tests, which then required me to understand the matter better. Slowly I began to understand why each of the monitoring tasks is required and started designing this aspect more elaborated.

During the project the client always came up with new requirements of which some exist in the old solution, but I was not aware of. Some of these requirements were covered by my as generic as possible engineering approaches in a satisfying way, but others that I implemented explicitely were not satisfying to the client at first.

For this reason it became important to do prototyping, which I started a little late.

Luckily, to some requirements, such as Performance, the implementation was obvious. Faster is better. I think I could clearly demonstrate some improvements and provide a nice strategy regarding this aspect for future development.

11 Outlook

SIMD-Processing seems to be very powerful. With Jacket a powerful framework exists. Because my intention was not to depend on additional commercial software, i did not investigate all options yet. However, should uncommercial solutions arise or Mathworks internal CUDA-framework become more powerful, then it is worth investigating. My rough guess is that the whole algorithm could perform faster on the GPU and would only require to transfer some scalars to the CPU in order for control-structures to decide how to proceed.

Also it may be interesting to extend batch processing to multiple computers. It is currently untested, but may work with minimal effort and I assume that the overhead is minimal, because algorithm execution remains isolated to a node and does not communicate using the network, if no monitors are applied. Behaviour of monitors in a cluster has to be looked at separately.

Since the solution now provides persistence and easy creation of configuration profiles, as well as simplified monitoring and batch processing, the next step from my point of view is to develop more sophisticated profile generators. In a sequential batch process these profile generators could apply fitness functions to a currently running algorithm instance and, based on the results, decide for different parameter values for the next algorithm runtime or even the currently running instance, maybe introducing a genetic meta-algorithm which configures the actual algorithm. But before such an algorithm is implemented it must be clarified whether by varying a single parameter only a local optimum is found or a global. If the optimum is global, which means that changing other parameters will not change the parameters own optimum value, then the perfect configuration can simply be achieved by altering every parameter in isolation.

If a local optimum is found, a global optimum can only be found by altering multiple parameters in combination, resulting in an exponential amount of required tests in relation to the amount of parameters. A genetic algorithm may make sense where algorithm configurations are being looked at as candidate solutions to be encoded into chromosomes[12]. The amount of tests then can be reduced by evolutionary selection of well-performing configurations.

12 Glossary

Term	Description
MPEG	Moving Picture Experts Group
Subversion	Tool to manage multiple versions of the same file or directory
Java	Strongly typed object oriented programming language with C like syntax
C++	An object oriented language with low level programming capabilities like C.
CUDA	A framework delivered by NVIDIA to run calculations on the GPU using the advantages of SIMD-processing.
GPU	Graphics Processing Unit
SIMD	Single Instruction Multiple Data (a type of parallel computing). This is the case, when the user defines one instruction, which is executed on data set, on each element in isolation.
OpenCL	A programming platform and language for various types of processors (CPU, GPU).
Jacket	A third-party library for matlab interfacing with CUDA
VSJ	Visualization Society of Japan

Bibliography

1: Berthold K.P. Horn, Brian G. Schunk, Determining Optical Flow, 1981
2: Wikipedia, Declarative programming, 2011, http://en.wikipedia.org/wiki/Declarative_programming
3: Wikipedia, Functional programming, 2011, http://en.wikipedia.org/wiki/Functional_programming
4: Dave Foti, Inside MATLAB Objects in R2008a, 2008, http://www.mathworks.com/company/newsletters/digest/2008/sept/matlab-objects.html
5: MathWorks, Execute code loop in parallel, 2011, http://www.mathworks.com/help/toolbox/distcomp/parfor.html
6: AccelerEyes, Jacket Basics, 2011, http://wiki.accelereyes.com/wiki/index.php/Jacket_Basics
7: AccelerEyes, Jacket SLA, 2011, http://wiki.accelereyes.com/wiki/index.php/Jacket_SLA
8: MathWorks, MATLAB GPU Computing with NVIDIA CUDA-Enabled GPUs, 2010, http://www.mathworks.com/discovery/matlab-gpu.html
9: AccelerEyes, GDOUBLE, 2011, http://wiki.accelereyes.com/wiki/index.php/GDOUBLE
10: AccelerEyes, GEVAL, 2011, http://wiki.accelereyes.com/wiki/index.php/GEVAL
11: AccelerEyes, GSYNC, 2011, http://wiki.accelereyes.com/wiki/index.php/GSYNC
12: Wikipedia, Genetic algorithm, 2011, http://en.wikipedia.org/wiki/Genetic_algorithm