Model Driven Paediatric European Digital Repository

Call identifier: FP7-ICT-2011-9 - **Grant agreement no**: 600932
Thematic Priority: ICT - ICT-2011.5.2: Virtual Physiological Human

Deliverable 14.3
Beta version Infrastructure Deployment Report

Due date of delivery: 29-02-2016
Actual submission date: 31-03-16

Start of the project: 1st March 2013
Ending Date: 28th February 2017

Partner responsible for this deliverable: MAAT
Version: 0.1

Dissemination Level: Public

Document Classification

Title	Beta version Infrastructure Deployment Report
Deliverable	14.3
Reporting Period	3
Authors	Sebastien Gaspard
Work Package	14
Security	PU
Nature	RE
Keyword(s)	Beta version Infrastructure Deployment Report

Document History

Name	Remark	Version	Date
Deliverable 14.3		0.1	03/02/2016
Deliverable 14.3		0.2	01/03/2016

List of Contributors

Name	Affiliation
Sebastien Gaspard	MAAT
Lucian Itu	UTVB
Jérome Revillard	MAAT
David Manset	MAAT
Harry Dimitropoulos	ATHENA
Emilie Pasche	HES-SO

List of reviewers

Name	Affiliation
Harry Dimitropoulos	ATHENA
Bruno Dallapiccola	OPBG

Abbreviations

DCV	Data Curation and Validation
ETL	Extract Transform Load
MG	Multigrid Method
GPUs	Graphics Processing Units
GUI	Graphical User Interface
One–Versus–All (OVA)	OVA
PCG	Preconditioned Conjugate Gradient method
SVM	Support Vector Machine
UDF	User-Defined Function
CDR	Clinical Data Repository
CBR	Case-Based Retrieval

TABLE OF CONTENTS

1 Project summary

MD-Paedigree is a clinically-led VPH project that addresses both the first and the second actions of part B of Objective ICT-2011.5.2:

1. it enhances existing disease models stemming from former EC-funded research (Health-e-Child and Sim-e-Child) and from industry and academia, by developing robust and reusable multi-scale models for more predictive, individualised, effective and safer healthcare in several disease areas;
2. it builds on the eHealth platform already developed for Health-e-Child and Sim-e-Child to establish a worldwide advanced paediatric digital repository.

Integrating the point of care through state-of-the-art and fast response interfaces, MD-Paedigree services a broad range of off-the-shelf models and simulations to support physicians and clinical researchers in their daily work. MD-Paedigree vertically integrates data, information and knowledge of incoming patients, in participating hospitals from across Europe and the USA, and provides innovative tools to define new workflows of models towards personalised predictive medicine. Conceived of as a part of the "VPH Infostructure" described in the ARGOS, MD-Paedigree encompasses a set of services for storage, sharing, similarity search, outcome analysis, risk stratification, and personalised decision support in paediatrics within its innovative model-driven data and workflow-based digital repository. As a specific implementation of the VPH-Share project, MD-Paedigree fully interoperates with it. It has the ambition to be the dominant tool within its purview. MD-Paedigree integrates methodological approaches from the targeted specialties and consequently analyses biomedical data derived from a multiplicity of heterogeneous sources (from clinical, genetic and metagenomic analysis, to MRI and US image analytics, to haemodynamic, to real-time processing of musculoskeletal parameters and fibres biomechanical data, and others), as well as specialised biomechanical and imaging VPH simulation models.

2 Executive summary

As an update of D14.2, this document will just present the novelties of the infostructure. To have an exhaustive view of the platform, please refer to D14.2.

This document will first present the new dimension of the repository which is managing access rights, defining conceptually and technically the new abilities of the system. Then, the GUIs offering new functionalities and the integration of HES-SO CBR will be described, followed by the web version of Athena DCV and AITON, respectively managing curation and validation and statistical models.

Following the usual order since D14.1, work on GPUs will end this presentation just after a presentation of the anonymiser tools that gnúbila has provided to the project.

We remind the reader that all these components are coming in addition to the tools presented in deliverables D14.1 and D14.2 (we chose not to present them a third time in this document) and that this document does not by itself transcribe the entirety of the amount of work provided during the project's first three years.

3 FedEHR a Clinical Data Repository (CDR)

3.1 Hardware Architecture

3.1.1 Current installed node

Current architecture is composed of:

- 1 Node at OPBG Rome
- 1 Node at CCPM Taormina
- 1 Node at DHZB
- 1 Portal
- 1 Central Server

The existing installation provides one additional node at DHZB gained from Cardioproof collaboration.
The nodes are currently installed and connected together through a FastWeb secured connection. This allows all sites to share information with ease.

3.1.2 Planned node

Added to the current installation, and despite the unavailibity of funds to cover the cost, KUL has provided a hardware that is currently under installation to add a new node.
At the time beeing, GOSH is also planning to acquire hardware to provide an access point.

3.2 Patient Centric model

As described in the deliverable D14.2, the current description of FedEHR architecture provides an evolutionary structure of data starting from the patient.

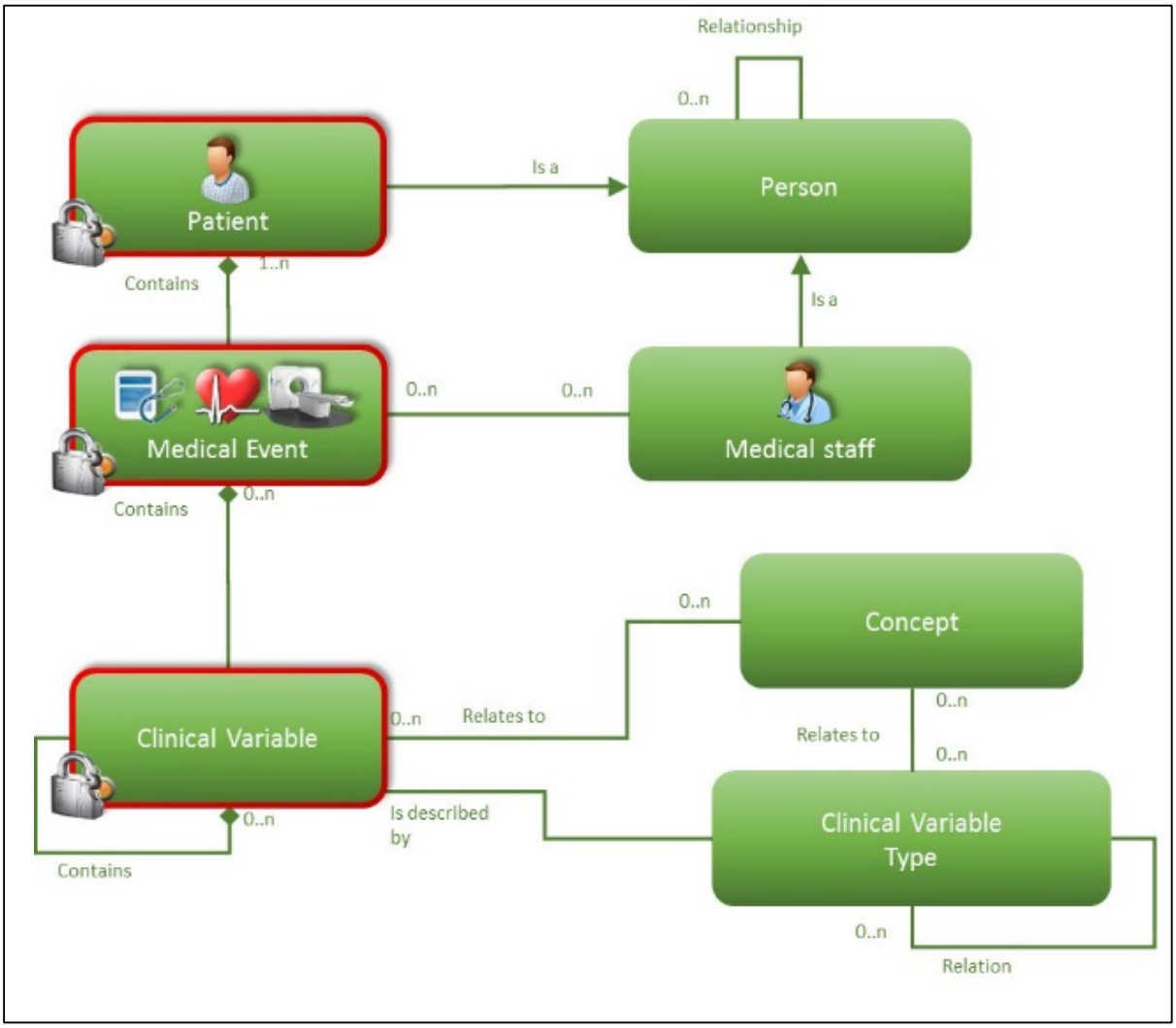

Figure 1: Patient-Centric Model

The FedEHR metamodel is by design easily evolvable and conceptually fully compatible with health modelling tools. Designed to be simple and understandable by physicians and non-IT experts, FedEHR thus offers a simple, powerful and harmonized patient-centric representation of EHR data.

The core system model can be synthetized as follows (for the sake of clarity, only the main objects are represented here).

The model revolves around the Patient concept. It is a global pattern for all medical information types. A Medical Event can relate to one or several Patient(s) and Medical Staff. A Medical event is composed of a set of Clinical Variables, each of which can be a simple value (i.e., blob, coded value, integer, float, measurement, etc.) or a more complex data (i.e., tree of values, reference to a complex document stored in an external document or image repository, etc.).

A main concept called Clinical Variable Type is attached to Clinical Variable, which ensures semantic consistency, and thus allows concepts from external terminologies such as HL7, SNOMED, ICD-10 (or any other) to be linked in to any clinical variable or clinical variable type. This way, data can be turned meaningful while the original patient-centric model remains simple enough to tackle any possible types of data faced in targeted health information systems.

3.3 FedEHR Data Distribution & Access Strategies

FedEHR allows for various data distribution strategies, useful to best-fit geographically dispersed and heterogeneous data ecosystems. In FedEHR, data schemata are represented as trees, which can be physically partially or fully stored in different backend instances. The following figure depicts the four possible distribution strategies.

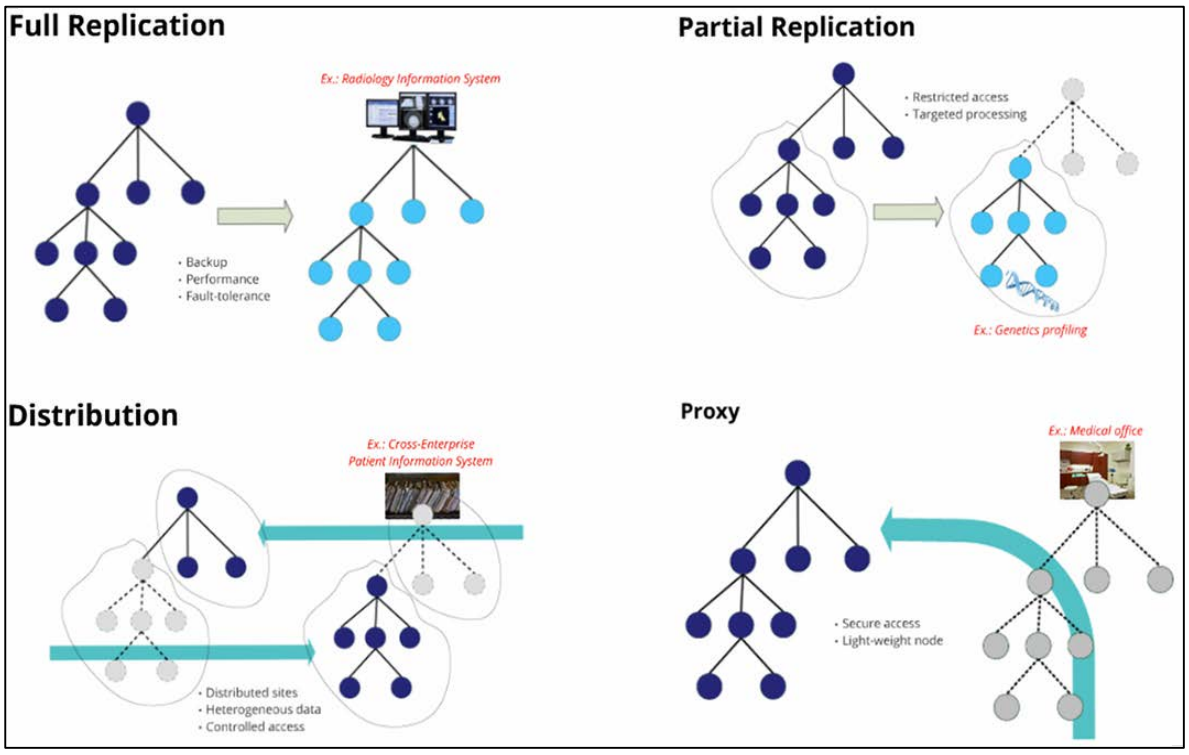

Figure 2: Data Distribution

With full replication, top-left of the figure above, a complete database can be replicated between several sites, thus avoiding single point of failure. This can be useful to RIS systems (Radiology Information Systems) where a high quality of service is needed, in terms of access performance and load-balancing. Partial replication, top-right of the figure, has the advantage of providing good scalability as data can be organized in sub-trees, each of which is managed independently, in and by concerned sites. A federation of individual schemata, bottom-left of the figure, is the most interesting solution in the present case, where manipulated data can potentially be highly sensitive. The latter ensures data collection, ownership and control remain local, while providing harmonized access to authorized users. In this particular scheme, users navigate a global data catalogue, whereas access to the data stored in individual sites is strictly regulated by the site itself and according to local data protection rules. Finally, also worth noticing, a variant of FedEHR's federation strategy is the federation with proxy redirection, bottom-right of the figure. Federation with proxy redirection allows sites not equipped with IT to contribute data to an existing database. This can be of interest in the case of isolated medical offices, for instance.

Syntactically speaking, FedEHR also provides connectors to mostly utilized database backend (from MySQL,[1] to PostgreSQL,[2] to Oracle,[3] SQLite,[4] and Berkeley DB XML[5]) and offers a SQL (Structured Query Language)-like query interface and transactional engine, thus harmonizing and strengthening querying over heterogeneous data sources, irrespective of their locations and conforming to data confidentiality requirements.

[1] MySQL is the world's second most widely used open-source relational database management system. It is named after co-founder Michael Widenius's daughter, My.

[2] PostgreSQL, often simply Postgres, is an open source object-relational database management system with an emphasis on extensibility and standards compliance.

[3] The Oracle Database (commonly referred simply as Oracle) is an object-relational database management system produced and marketed by Oracle Corporation: www.oracle.com.

[4] SQLite is a relational database management system contained in a small C programming library. In contrast to other database management systems, it is not a separate process that is accessed from the client application.

[5] Berkley Database engine EXtensible Markup Language

3.4 FedEHR Data access rights management

3.4.1 Model

Taking advantages from the simple data model and distribution abilities, FedEHR implements a simple but powerful Role Based Access Control (RBAC). Inspired from the literature and designed to respond to GDPR[6] specifications, the model has been enriched by a fine tuned access controlled piloted by the Data Protection Officer (DPO) of each centre.

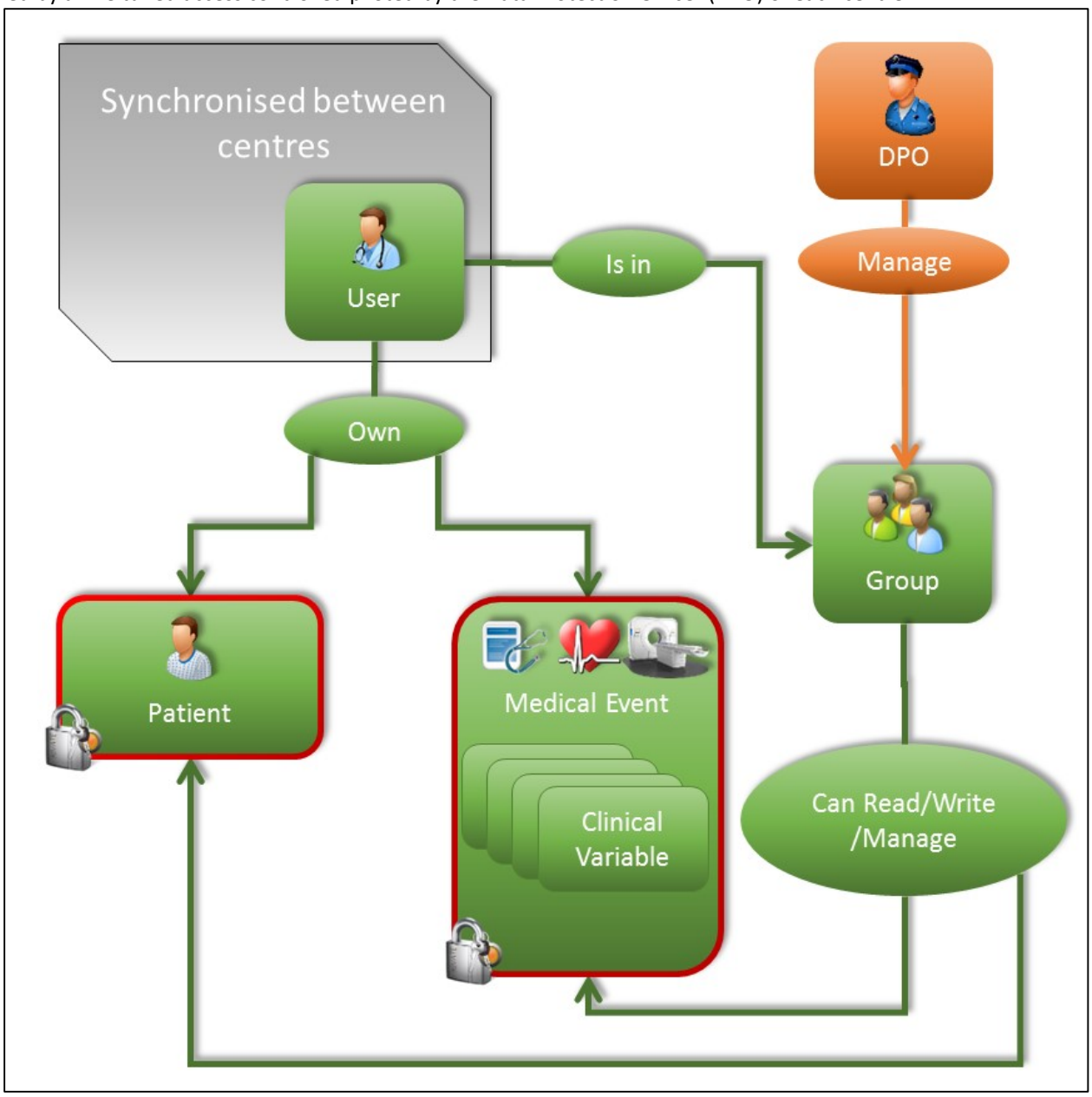

Figure 3: Access Rights Management

Users are registered in a replicated table managed by the central node manging synchronisation between the different centres of the shared system. This allows a shared view of users. For each centre DPOs can create groups referencing both user from their local centre and external users.

[6] GDPR : General Data Protection Regulation
http://www.europarl.europa.eu/meetdocs/2014_2019/plmrep/AUTRES_INSTITUTIONS/COMM/COM/2015/12-17/COM_COM%282012%290011_EN.pdf

Groups represent the concept of "group of users". Each group can be composed of a subset of users and reference a set of Access Rights.
Access Rights defines the ability of groups on restricted objects. They are incremental and are giving rights as following:

- No Right: the concerned object cannot be viewed by the user; it cannot even be listed.
- Read: The user can see the restricted object.
    - In the case of a patient, the user can see the patient available identity information, address, hospital and the mist of Medical Event IDs attached
    - In the case of a Medical Event, the user has access to the whole information of the event but not of the patient
- Write: The user has the right to add and modify (when relevant) information he can see.
- Manage: This access allows user to give access rights to groups

3.4.2 Behaviour

Users register themselves at central node (on the shared portal) using a simple easy to understand wizard and get an account. The registration can be directly onto the central node or delegated to a specific authentication authority (like EDUGAIN, Facebook or LinkedIn) depending on the degree of security and project policies.

The connection does not give any access to the data. The registration makes identities available to the different DPOs to be added to groups.

DPOs manage groups at the centre level, despite being centralised, users are assigned to local groups.
DPOs can define as many group as they want to create.
DPOs can manage all patients and medical events access rights.

3.4.3 Java/WebService API

3.4.3.1 Objects

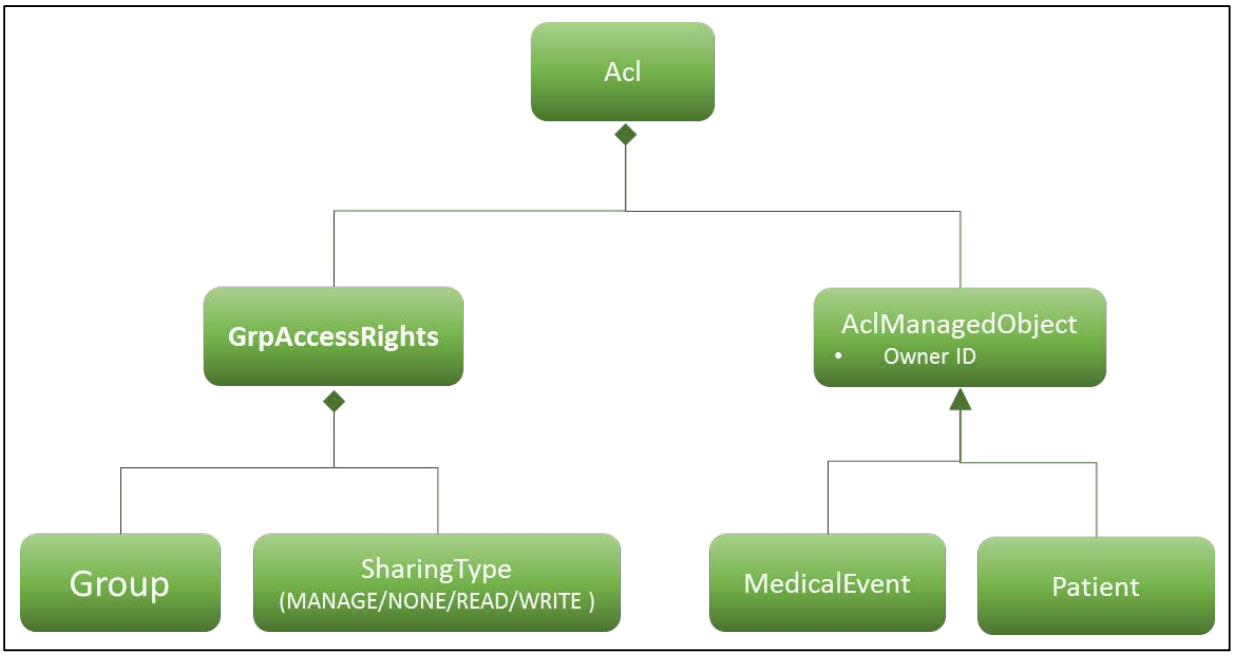

Figure 4: Access rights Objects

Acl is composed of a grpAccessRight and a AclManagedObject. This object is used to get/set information about access rights.

GrpAccessRights is composed of a reference to a Group and a Sharing Type

Group represents the logical concept of groups, groups are managed by the DPO, have a name and references a list of users acquiring all the GrpAccessRights the group have.

SharingType is the object that represents the ability of the group for one AclManagedObject. *Read*/*Write* and *Manage* are stored in the DB, *None* is the default value and is only used to unset rights.

AclManagedObject is an abstract concept, it represents the own-ability of a object. Each AclManagedObject is created associated to an owner who is a physician that is responsible of the attached data.

MedicalEvent and **Patient** are AclManagedObjects, they encapsulate the patient data.

3.4.3.2 Implemented API

A java/WebbService API is available to use the access right management. This API exposes all the secured functions that are needed but one (DPO assignation). The functions are:

addGroup(Group)
what: add a group to the system
who: DPO

addGroups(Groups)
what: add a list of groups to the system
who: DPO

addUsers(Users)
what: add a group to the system
who: Imported from central Node

addUsersToGroup(GroupUsers)
what: assign Users to group
who: DPO

deleteGroup(Group)
what: remove a group from the system
who: DPO

deleteGroups(Groups)
what: remove a list of groups from the system
who: DPO

getSharing(AclManagedObjects)
what: give the list of access each object of the list have
who: Anybody with Read access to the objects

listACLs(QGroup)
what: give the list of objects the group have
who: Anybody with Read access to the objects

listGroups(QGroup)
what: lists groups matching the query
who: anyone

listGroupUsers(QGroup)
what: lists the users of a group
who: anyone

listUserGroups(QUser)
what: lists the groups of a user
who: anyone

listUsers(QUser)
what: lists matching the query
who: DPO

setSharing(Acls)
what: adds access rights to objects
who: DPO, owner of concerned objects or user in group with Manage right.

updatePatientsOwner(OwnerAndPatients)
what: assign a new owner
who: Owner

4 Case-based retrieval service

4.1 Text based

The Case-Based Retrieval (CBR) service aims to help physicians to find similar patients based on the clinical reports of a given patient. In addition, the service proposes a summary of the returned cases of similar patients at different points in time. In the system, the basic search item is the episode of care.

The CBR is developed by HES-SO. Figure 5 illustrates the global workflow of the CBR. On the HES-SO server, an index is created: data are extracted from the PCDR and indexed using a local instance of Apache Solr. A MeSH normalization of the clinical syntheses is locally performed and stored in the indexes. The graphical user interface, together with all dependent services, are located on the MD-Paedigree Portal. The services communicate with the index to obtain the similar cases list using Json exchange messages.

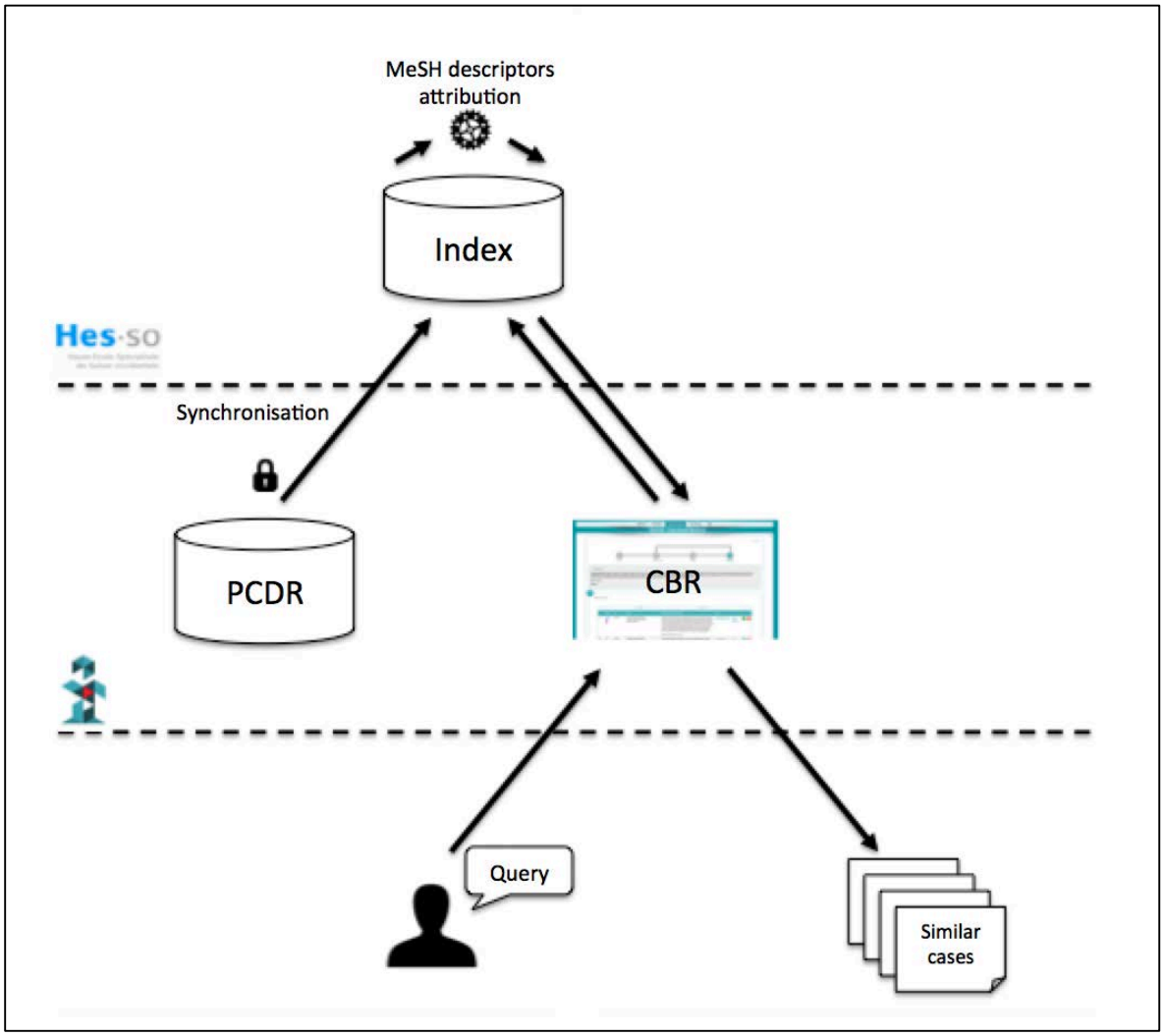

Figure 5: Overall workflow of the CBR engine.

Clinical data from the MD-Paedigree project are obtained through the secured PCDR API developed by gnúbila. HES-SO obtained a GRID certificate delivered by SwiNG (i.e. one of the certificate authorities delivering GRID certificates in Switzerland) in order to be trusted by the PCDR server. Thus, the global workflow includes a secured synchronization between HES-SO and the MD-Paedigree portal.

A set of 47,433 episodes of care was extracted, corresponding to 33,674 distinct patients. The following demographic information is extracted: gender, birth date, date of the episode of care, conclusion content (i.e. clinical synthesis). The data is

then directly indexed. In addition, a set of MeSH descriptors is automatically assigned to the clinical syntheses. These descriptors are also indexed.

In the current version of the CBR, and in order to facilitate the display in the CBR graphical user interface, the following information is stored in the Solr Index: age of the patient when the episode of care occurred, gender, clinical synthesis and MeSH descriptors. However, in the final version of the CBR, it is planned that the information will not be stored anymore in the index but directly fetched from the PCDR at query time.

Regarding the graphical user interface, a full integration of the CBR has been performed (Figure 6). The HES-SO team has been trained by the Gnùbila team, in order to be able to integrate the current version of the CBR and its future updates. An instance of the Liferay Portal has been locally installed (version 6.1.2) at HES-SO for sake of development. The current CBR *servlet* has been transformed to a *portlet*. Once ready, the HES-SO team pushed the final version on a web-based Git repository manager and the Gnùbila team deployed it on the MD-Paedigree portal.

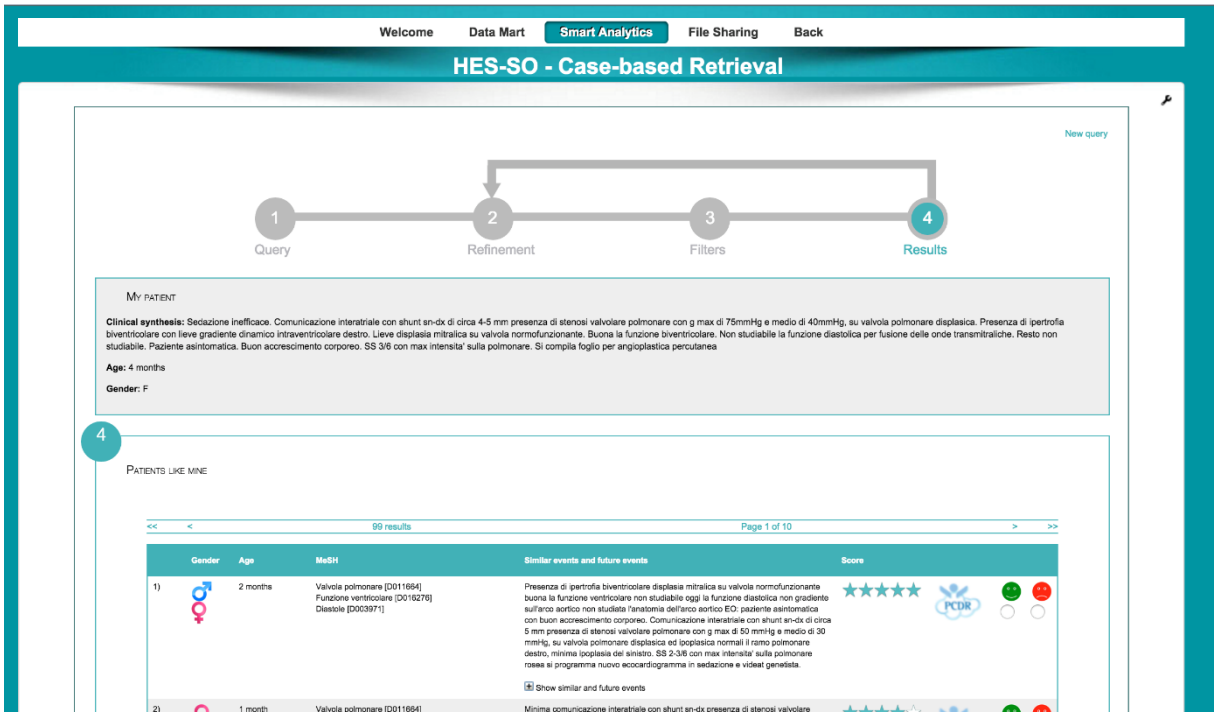

Figure 6: Fully-integrated GUI of the CBR

4.2 Image based

As part of the similar case retrieval it is planned to include also visual retrieval, so visual characteristics that are extracted directly from the image pixel information. Such content-based image retrieval has shown in the past to well complement the text-based or structured querying. In year 3 checks were done with the disease areas to select the best possible scenario and gather sufficient image data for training in at least one of the disease areas. Prototypes were developed on retrieval of images from the literature by visual means in year 3. In year 4 this will be extended to at least one of the disease areas and evaluation in combination with the semantic retrieval for a mutlimodal retrieval approach.

5 DCV : Data Curation and Validation

An updated version of the web-based Data Curation and Validation (DCV) tool has been released as part of the beta prototype of the infostructure. The DCV tool, developed by ATHENA, is a web application offering an advanced (semi)-automatic data cleaning process for MD-Paedigree data. The tool uses a client-server architecture, illustrated by the following figure.

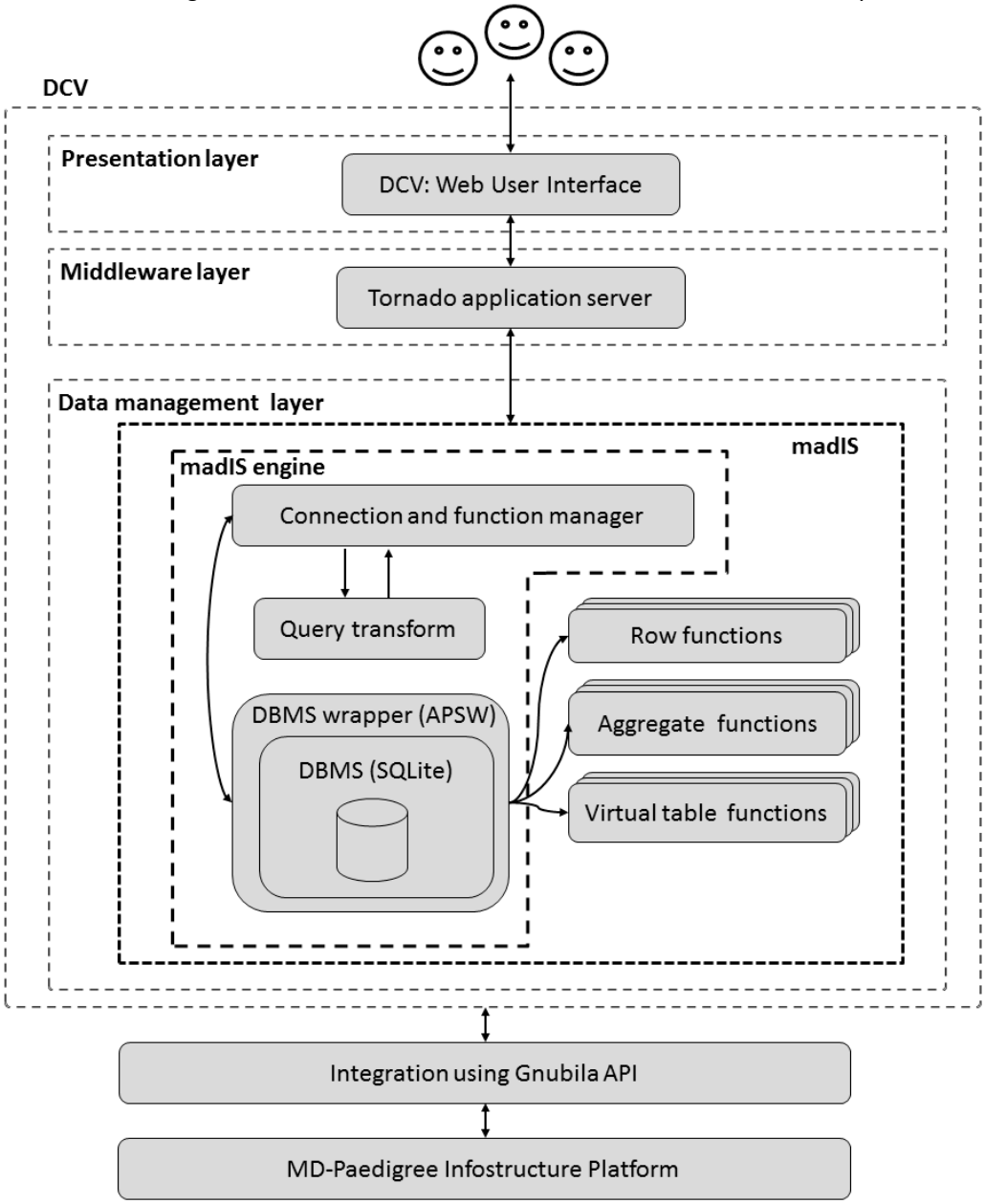

Figure 7: Architecture of the new and updated DCV tool. It is connected with the MD-Paedigree infrostruction via Gnubila's API

For more information about DCV's alpha implementation please consult deliverable "D15.2 - DCV curation tools and services to automatically and manually acquire high-quality curated data". For the beta release implementation and details on what has been improved and updated since the previous version, please also consult "D16.2 - Beta Prototype of KDD & Simulation platform".

6 AITON: Knowledge Discovery (KDD) and Simulation Platform

AITION is the information processing, knowledge discovery (KDD) and simulation platform for Big Data Healthcare, as well as, the related, well-defined KDD workflow that promotes model-guided personalized medicine. AITION is developed under WP16 and is made up of a number of different modules, as described in detail in deliverable "D16.1 - First report on biomedical knowledge discovery and simulation for model-guided personalized medicine". The beta release of the infostructure incorporates the components described below.

Under task "T16.1 - General data analysis and knowledge discovery tools", some well-established Machine Learning (ML) techniques and algorithms have been implemented on top of ATHENA's EXAREME (ex ADP/madIS) data flow processing system, following the same architecture used by the DCV tool (T15.1). This way, all platforms and tools developed by ATHENA will be integrated providing an end-to-end data pre-processing, data analysis and data mining platform *with only one point of integration with the MDP platform.*

More specifically, the current version of DCV mainly consists of data preprocessing and cleaning methods. In order to further extend its functionality and provide a more complete user experience, we have decided to incorporate several well-established ML algorithms within the same WEB-based data analysis platform, following the same architecture. This way, we are integrating together in a streamlined fashion all platforms and tools produced by ATHENA RC, providing an end-to-end online, data cleaning, pre-processing, data analysis and data mining platform, with only one point of integration with the MDP platform.

Thus, the end-user besides being able to pre-process data (e.g. detecting errors or outliers), will also have the opportunity to identify groups and similar cases or create models that predict the value of one or more target variables using the same platform. For this purpose, we have incorporated an open source Python library, scikits.learn (http://scikit-learn.org/stable/) which contains a number of well-established machine learning algorithms and techniques implemented in Python libraries. These libraries can be easily imported on top of EXAREME as specific User-Defined Funtions (UDFs) of the madIS system (EXAREME's worker). Such UDFs, construct predictive or clustering models, estimate new values for unlabeled data, and categorize new data samples.

At this point, three general UDFs (operators) have been developed:

- one for creating clustering (unsupervised) models,
- one for training the classification / regression (supervised) models, and
- one for predicting new values for unlabeled data samples.

Furthermore, as will also be described in detail in the forthcoming deliverable "D16.2 – First report on biomedical knowledge discovery and simulation for model-guided personalized medicine", we have already incorporated within DCV ten (10) new algorithms: 4 clustering, 3 supervised, and 3 methods for dimensionality reduction.

A screenshot of the new "KDD section" extending the capabilities of the DCV tool is shown in the following figure. In particular, the user can now choose among several extra preprocessing techniques like dimensionality reduction for visualization or feature extraction. The user can also identify similarities among the samples, or construct classification or regression models via the clustering & supervised sections respectively. All models are serialized and stored in a compressed file. We are also working to store the model in user's database but most of the time what is better in terms of memory allocation depends on the model's size. Therefore, each user will have the ability to choose a trained model in order to cluster or classify new incoming samples.

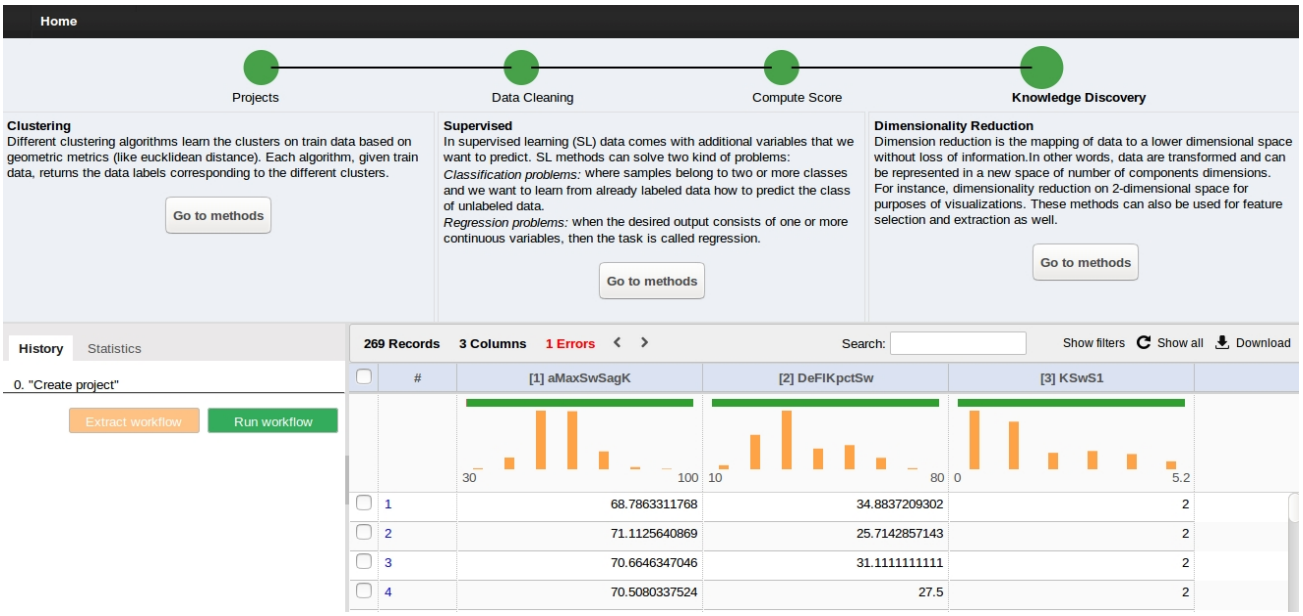

Figure 8: DCV - KDD integration: An extra flow-step ("Knowledge Discovery") has been added within the DCV tool giving the user the opportunity to explore new knowledge discovery techniques.

Under task T16.3, the AITION Desk tool (rich GUI application) has been adapted to MD-Paedigree related requirements and a first level integration (for data retrieval) with the platform has been achieved. In addition, we have re-implemented specific algorithms for Bayesian Network structure learning initially written in Matlab, following a Service Oriented Architecture that will give as the opportunity to provide such functionality on top of the MD-Paedigree platform.

Integration of the AITION/DCV tools with the platform: A first level integration (for data retrieval) with the platform has been achieved. More specifically, at this point we are able to first explore the underlying domain model (data types, model's hierarchy) and select specific variables. Then, based on these variables, the related SQL is generated to parse data through the MD-Paedigree platform API.

7 Gnubila Anonymiser

As a consequence of the amount of data that was not following the MD-Paedigree guidelines in term of anonymization and the lack of such a tool in different centres, gnúbila has decided to provide for free its anonymisation solution.
This anonymization tools is both installed as a standalone at some partners sites and also embedded inside all importers provided by gnúbila.

7.1 Architecture

The architecture of the Anonymizer is based on the Application Server Tomcat 7[7] and contains an ETL (Extract Transform Load) engine provided by Apache Camel[8]. Apache Camel is an open source Java framework that focuses on making integration easier and more accessible to developers. It does this by providing:

- concrete implementations of all the widely used Enterprise Integration Patterns (EIPs),
- connectivity to a great variety of transports and APIs,
- easy to use Domain Specific Languages (DSLs) to wire EIPs and transports together.

The FedEHR Anonymizer is generic by design and therefore configurable to respond to the latest and evolving data privacy regulations. The proposed version of the solution will be widely customizable through the privacy profiles, which data curators can define based on ethical concerns and applicable regulations.
The FedEHR Anonymizer is able to process different file types like Digital imaging and communications in medicine files (or DICOM), CSV files (Comma-Separated Values).
The architecture allows the Anonymizer to provide a wide flexibility regarding the protocols for the input/output/quarantine management (local file, FTP/SFTP, PACS, etc.).
The Anonymizer provides a Web Console, based on Hawtio[9], to facilitate the administration of the Camel routes. The Camel plugin of the FedEHR Anonymizer Web Console allows getting statistics and graphical charts about the anonymization process, but also more importantly a graphical visualization of your Camel Routes that will help you to customize your routes.

7.2 FedEHR Anonymizer Index Database

The FedEHR Anonymizer Index Database is based on the concept of Master Patient Index (MPI). A Master Patient Index (MPI) is an electronic medical database. The MPI stores and maintains a unique index (or identifier) with different information about the patient (patient name, gender, date of birth, etc.), and it can also include data on physicians or other medical staff.
The Master Patient Index ensures that each patient is stored only once within all the system and maintains all the data consistent.
The FedEHR Anonymizer is able to dump any identifying data extracted during the anonymization process into a Master Patient Index, called FedEHR Anonymizer Index Database, or Index Database. This MPI, based on MapDB[10], is an embedded database engine.

7.3 Anonymize files

Once started, to invoke the Anonymizer with the local route, you have to paste your original file into the Input directory (specified during the installation step, see above).
The .camelLock file indicates that your file (Dicom/CSV/...) is being processed by the Anonymizer:

Figure 9: Processing a DICOM

[7] Apache Tomcat: https://tomcat.apache.org/index.html
[8] Apache Camel: http://camel.apache.org/
[9] Hawtio: http://hawt.io/
[10] MapDB: http://www.mapdb.org/

Once your file is successfully processed, your file is moved into the output directory.
If an error occurred during the anonymization process, the original file is moved into the quarantine folder (configured above).
A report containing the error message and the stacktrace is also created:

Figure 10: Quarantine folder

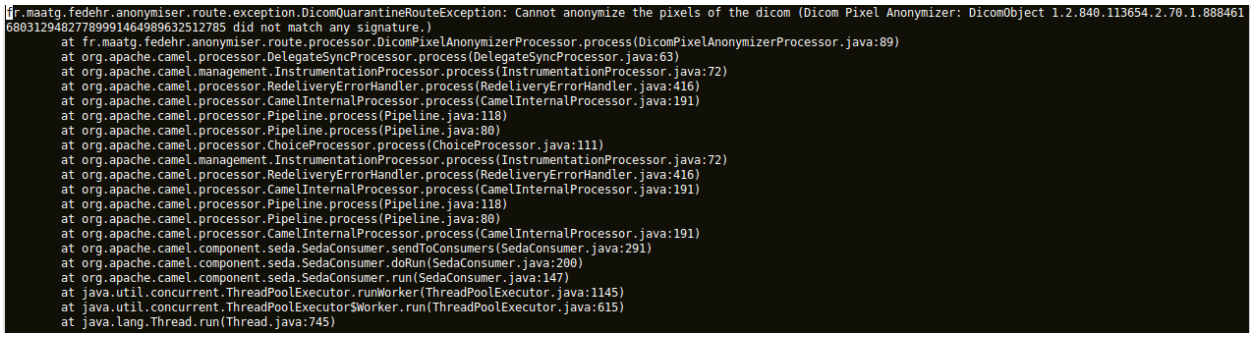

Figure 11: Quarantine report

7.4 Manage the Camel routes

The FedEHR Anonymizer provides a graphical tool, the FedEHR Anonymizer Web Console to easily manage and customize the Camel routes.

7.4.1 Access to the graphical Camel environment

- To access the graphical Camel environment, the User must be uncommented and the Administrator Password must be changed in the *TOMCAT_HOME/conf/tomcat-users.xml* configuration file.
- Open your favourite Web Browser to the following *TOMCAT_ADDRESS/hawtio.*
  1. Use the login and password that you customized above to log in to the FedEHR Anonymizer Web Console.

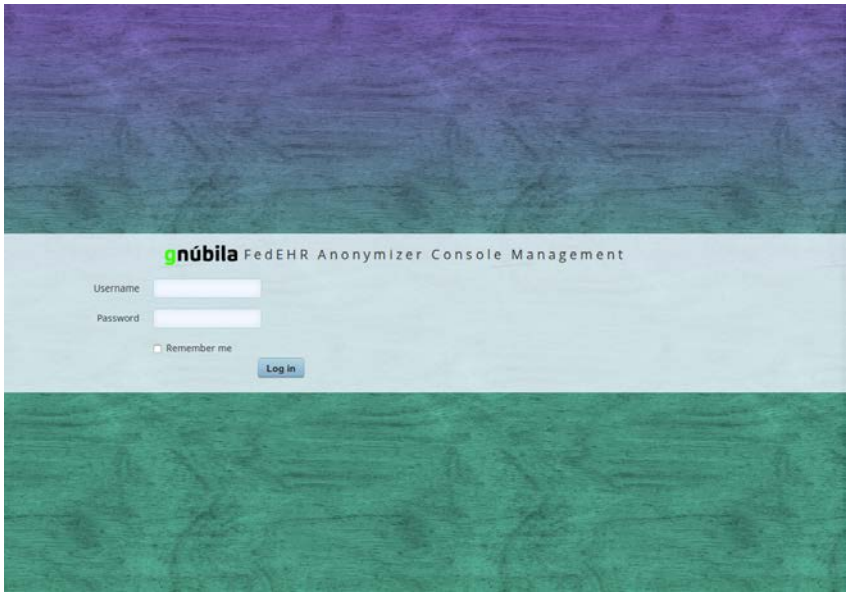

Figure 12: FedEHR Web Console: Login page

2. Go to the Camel tab.

7.4.2 Visualize and Monitor your routes

The FedEHR Anonymizer Web Console allows you to visualize and monitor your Camel routes. It's a useful feature to deal with complex Camel routes.

1. The different routes registered in the Camel Context are listed in the left part of the page (Figure 13).

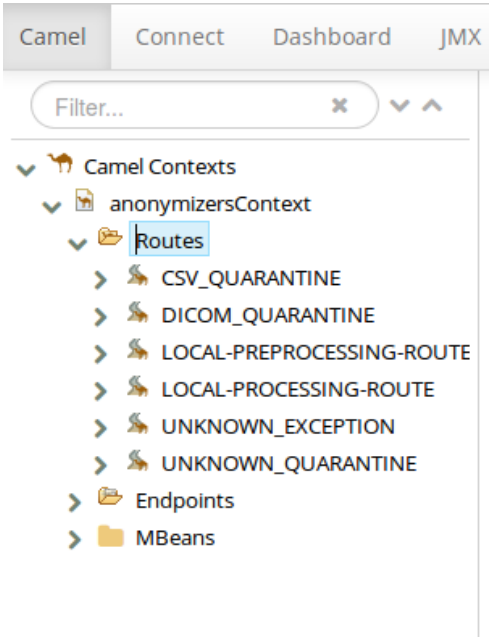

Figure 13: Registered Routes in the Camel context

2. You can get real time information (Figure 14) about the status of your routes.

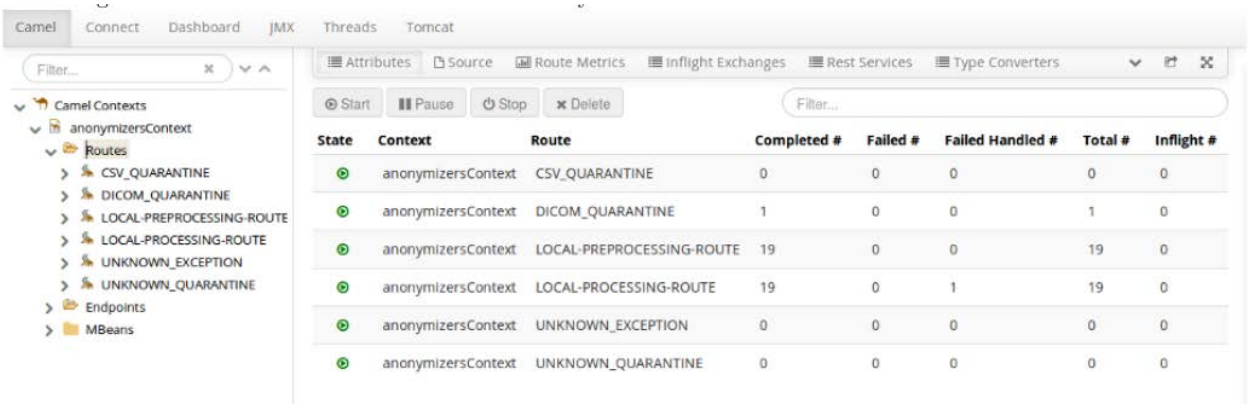

Figure 14: Real time information about the Camel routes

3. To graphically visualize your routes (Figure 15), select a route and choose "Route Diagram"

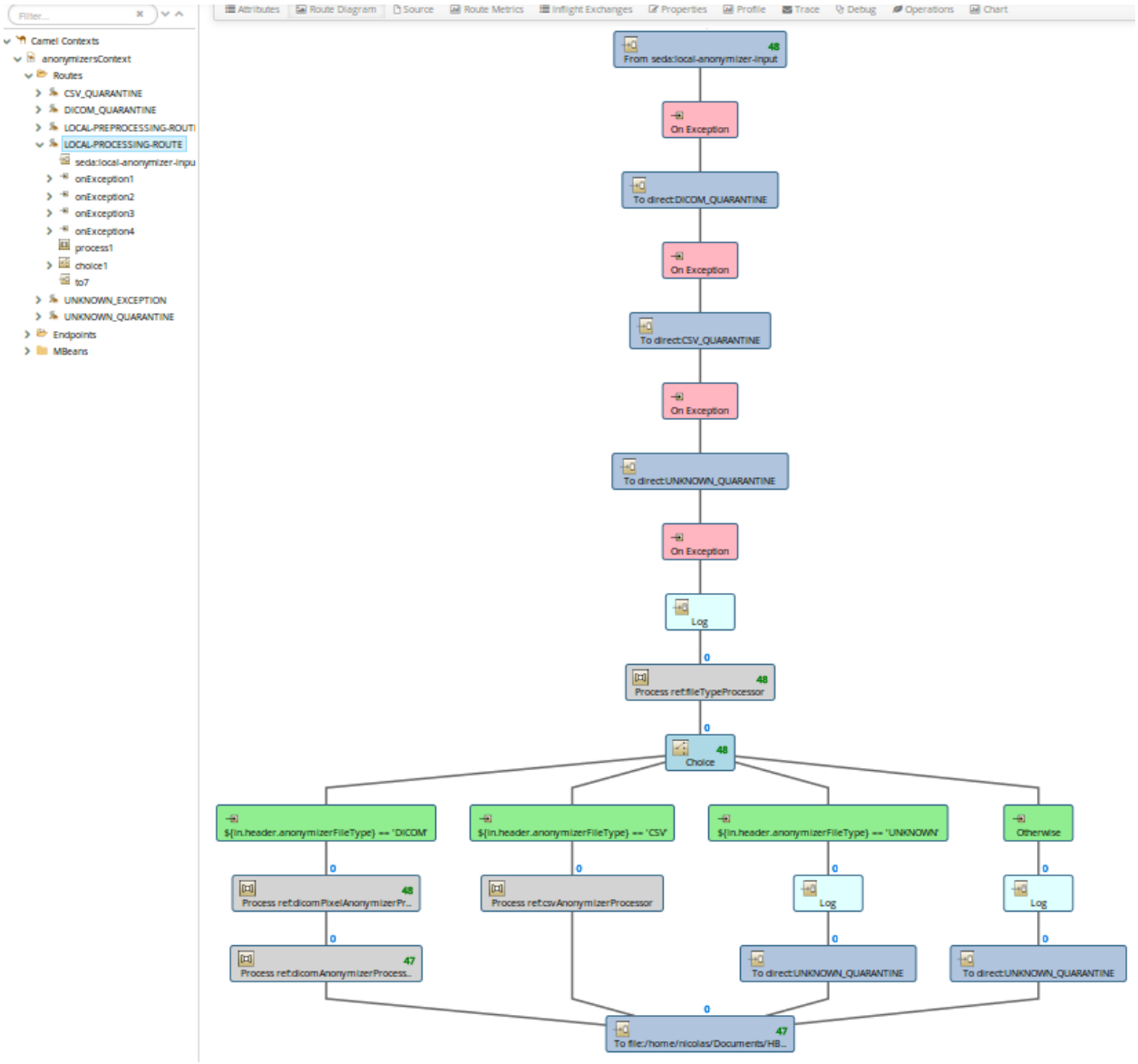

Figure 15: Graphical representation of a Camel route

7.4.3 Graphically customize Camel routes

The FedEHR Anonymizer Web Console allows the administrator to modify on the fly the Camel routes (Figure 16). In order to do this:

1. Select a route on the left side of the interface.
2. Click on Source tab.
3. Process your modification and validate the modification by clicking on Update.

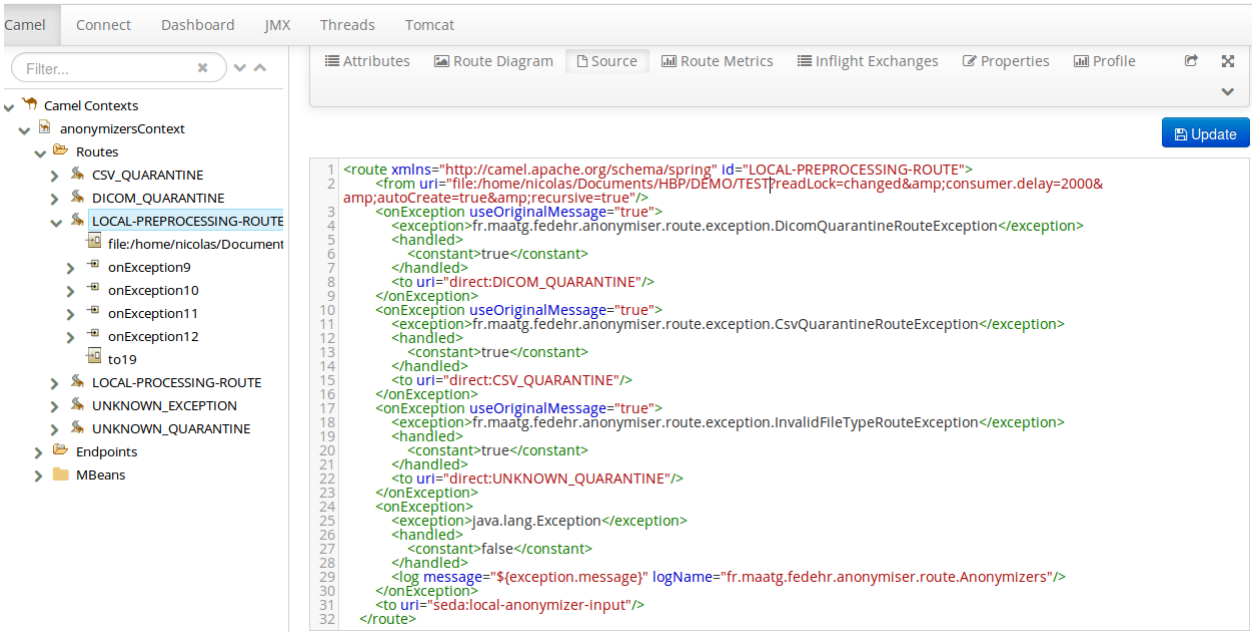

Figure 16: Graphical customization of a Camel route

8 GPU based processing and computation

8.1 GPU accelerated geometric multigrid method

8.1.1 Introduction

The two most popular algorithms for the solution of the sparse linear systems of equations resulting from the discretization of partial differential equations are the preconditioned conjugate gradient method (PCG) [Gui et al., 2012] and the multigrid method (MG) [Briggs et al., 2000], [Trottenberg et al., 2000], in its two variants geometric MG (GMG) and algebraic MG (AMG). The PCG method is regularly used for solving sparse symmetric positive definite linear systems, it is easy to implement, and converges in at most n steps to the solution (n is the size of the system) [Ament et al., 2010].
Originally, multigrid methods were developed to solve boundary value problems posed on spatial domains. More recently, the original multigrid approach has been abstracted to problems in which the grids have been replaced by more general levels of organization [Briggs et al., 2000]. The multigrid method is based on a hierarchy of discretization levels, whereas the corrections performed at the coarser discretization levels improve the convergence rate of the solution on the finest discretization level. The GMG method requires specific information on the hierarchy of discretizations, but, if this information is available, it is considerably more efficient than the AMG method [Ruge et al., 1987].
Previous researches have demonstrated that the GPU-based implementation of the GMG outperforms its CPU-based counterpart. Hence, in the current activity we have focused on a more in-depth analysis of the GPU-based GMG algorithm. Specifically we employed different GMG variants, different discretization schemes for the Poisson equation, varying number of smoothing steps during restriction and prolongation, we used single and double precision computations, and different discretization resolutions. Finally, we determined the performance gap between the GMG method and the PCG method on a state-of-the-art GPU.

8.1.2 Methods and implementation

To study the performance of the GMG method we consider the Poisson equation. To address the aspects mentioned in the introduction, in the following we specifically refer to the steady-state heat conduction problem and apply a finite difference method for its discretization in a three-dimensional domain. A uniform mesh of points is used, and, by applying central differencing, three different discretization schemes are considered, leading to:
a 7-point stencil,

$$T^{n}_{i,j,k} + T^{n}_{i+1,j,k} + T^{n}_{i-1,j,k} + T^{n}_{i,j+1,k} + T^{n}_{i,j-1,k} + T^{n}_{i,j,k+1} + T^{n}_{i,j,k-1} - 7T^{n}_{i,j,k} = 0$$

a 19-point stencil,

$$T^{n}_{i,j,k-1} + T^{n}_{i-1,j,k-1} + T^{n}_{i+1,j,k-1} + T^{n}_{i,j-1,k-1} + T^{n}_{i,j+1,k-1} + T^{n}_{i-1,j,k} + T^{n}_{i+1,j,k} + T^{n}_{i,j-1,k} + T^{n}_{i,j+1,k} + T^{n}_{i-1,j-1,k} + T^{n}_{i-1,j+1,k} + T^{n}_{i+1,j-1,k} + T^{n}_{i+1,j+1,k} + T^{n}_{i,j,k+1} + T^{n}_{i-1,j,k+1} + T^{n}_{i+1,j,k+1} + T^{n}_{i,j-1,k+1} + T^{n}_{i,j+1,k+1} - 19T^{n}_{i,j,k} = 0$$

and a 27-point stencil,

$$T^{n}_{i,j,k-1} + T^{n}_{i-1,j,k-1} + T^{n}_{i+1,j,k-1} + T^{n}_{i,j-1,k-1} + T^{n}_{i,j+1,k-1} + T^{n}_{i-1,j-1,k-1} + T^{n}_{i-1,j+1,k-1} + T^{n}_{i+1,j-1,k-1} + T^{n}_{i+1,j+1,k-1} + T^{n}_{i-1,j,k} + T^{n}_{i+1,j,k} + T^{n}_{i,j-1,k} + T^{n}_{i,j+1,k} + T^{n}_{i-1,j-1,k} + T^{n}_{i-1,j+1,k} + T^{n}_{i+1,j-1,k} + T^{n}_{i+1,j+1,k} + T^{n}_{i,j,k+1} + T^{n}_{i-1,j,k+1} + T^{n}_{i+1,j,k+1} + T^{n}_{i,j-1,k+1} + T^{n}_{i,j+1,k+1} + T^{n}_{i-1,j-1,k+1} + T^{n}_{i-1,j+1,k+1} + T^{n}_{i+1,j-1,k+1} + T^{n}_{i+1,j+1,k+1} - 27T^{n}_{i,j,k} = 0.$$

As described in section 8.1.1, geometric multigrid methods (GMG) refer to a group of algorithms for solving differential equations using a hierarchy of discretizations (Figure 17). The discretization is applied for different grids, whereas the grids have successively larger spacing between the nodes. All GMG variants are based on successive transitions from fine to coarse grids and back. Hence, the basic steps of the GMG method are:

- **relaxation** (smoothing): a simple iterative method like Jacobi or Gauss-Seidel is used to reduce the high frequency errors in the solution;
- **restriction**: the residual determined on a finer grid is downsampled to a coarser grid;
- **prolongation**: the residual on a finer grid is determined by interpolating the values from the coarser grid.

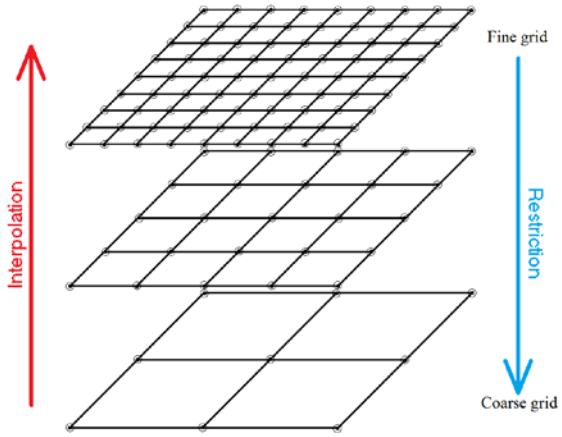

Figure 17: Basic concept of the geometric multigrid method: the solution is iterated through different discretization levels

The relaxation methods employed herein are red-black Gauss-Seidel (RBGS) for the 7-point stencil [Vizitiu et al., 2014], and Jacobi for the 19-point and 27-point stencils [Chung, 2010]. The red-black Gauss-Seidel method requires one array for storing the values, but the computations are divided into two sequential steps: grid nodes are marked as being red or black, whereas all neighbors of a node have the opposite color of the current node. Hence, when updating the values of the red nodes only values at black nodes are used, and vice-versa. The Jacobi method uses only values from the previous iteration and hence only one computation step is required at each iteration, but two different arrays are allocated for storing the previous and the current values at the grid nodes.
The GMG variants considered herein are displayed in Figure 18: V-cycle, W-cycle and full MG (FMG) scheme. Each figure depicts the strategy for a single iteration (multiple iterations are required to converge to the final solution).
Whereas GMG is based on an explicit solution scheme, the PCG method employs an implicit solution scheme for solving sparse linear systems of the form.

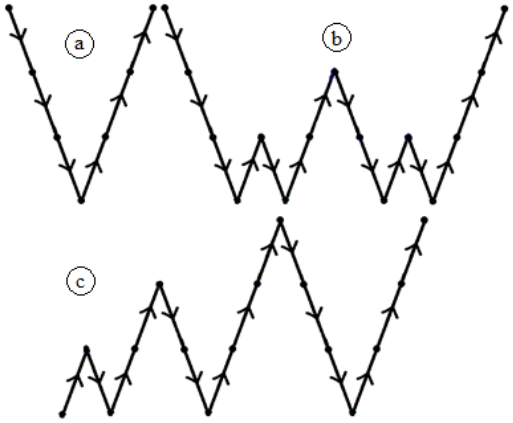

Figure 18: Geometric Multigrid variants: (a) V-cycle, (b) W-cycle, (c) Full MG (FMG).

The implementation of the V-cycle and W-cycle GMG variants are based on the μ-Cycle algorithm (**Algorithm 1**), which is a recursive scheme. The only difference is given by the parameter μ, which dictates how many times a new function will be launched: for the V-cycle μ = 1, whereas for the W-cycle μ = 2. When it is first launched, the algorithm starts at level 0, and, every time a new function is launched, a coarser grid is used [Briggs et al., 2000]. The values n1, n2, n3 determine the number of smoothing steps on the descending branch, at the coarsest level, and respectively on the ascending branch. Additionally to the prolongation step, on the ascending branch, a correction is employed: the values on the destination level are corrected based on the interpolated values computed from the source level (a matrix-sum operation is performed).

Algorithm 1 μ-Cycle

```
μ-Cycle(level)
    if( level is coarsestLevel )
```

apply n_2 ***smoothing*** steps
else
apply n_1 ***smoothing*** steps
compute residual
restrict to a coarser grid
μ-Cycle(level+1) μ **times**
prolongate
correct
apply n_3 ***smoothing*** steps
end

The multigrid method requires one storage array for each level (level 0 uses the largest array and occupies most of the execution time). At the coarsest level (level L) a 3D grid with 3x3x3 nodes is used. We consider Dirichlet boundary conditions, and, hence, the values on all faces of the domain are known (we set the values on five faces to 0, and one face to a non-zero value).

8.1.3 Results

To evaluate the performance of the GPU based GMG implementation we used a NVIDIA GeForce GTX Titan Black graphics card, and the CUDA toolkit version 6.0. The steady-state heat conduction problem was solved on a rectangular domain, and the Dirichlet boundary conditions, were set to 0°C for five facets and to 100°C for the remaining facet. The numerical solution was obtained on a grid of 129x129x129 nodes. Different numbers of smoothing steps were considered at different levels of the GMG method. Each configuration is described by a three-figure number: the first value determines the smoothing steps while restricting the grid, the second value determines the smoothing steps at the coarsest level, while the third number determines the smoothing steps while prolongating. All computations are performed in double precision and use the 7-point stencil when not otherwise stated, and iterations are performed until the average residual value no longer decreases from one iteration to the next (a value close to the limit of the corresponding floating point representation limit is reached).
First, we compare the different GMG schemes (V-cycle, W-cycle and FMG) in a 313 configuration with red-black Gauss-Seidel smoother. Figure 19 displays the dependence between the execution time and the average residual. The V-cycle scheme performs best: although it requires more iterations than the W and FMG schemes (13 iterations for V, 8 iterations for W, 11 iterations for FMG), the average residual decreases to 1e-14 in the shortest amount of time. Hence, for the following steps we present results for the V-cycle scheme.

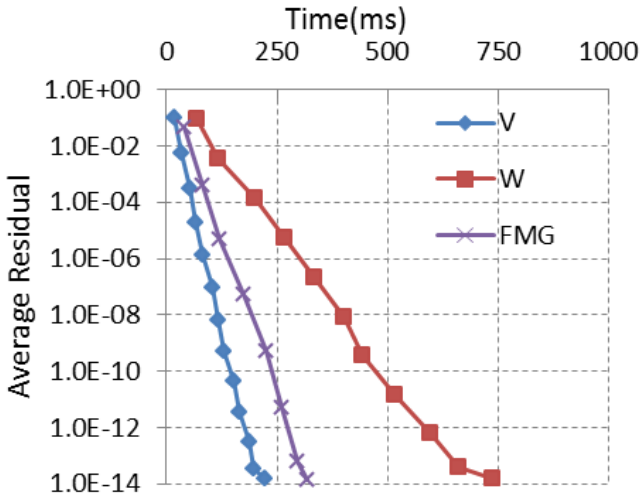

Figure 19: Comparison of different GMG schemes (V, W, FMG) when the RBGS smoother is used

Next, we analyze the effect of the smoothing configuration (for a RBGS smoother). Figure 19 displays the four best performing configurations: two or three smoothing steps are required during restriction and prolongation, while only one smoothing step

is required at the coarsest level. From the four depicted strategies, 212 and 312 perform best: 212 is slightly faster but requires one more iteration to reach an average residual of 1e-14 (11 vs 10 iterations), leading to approximately the same execution time.

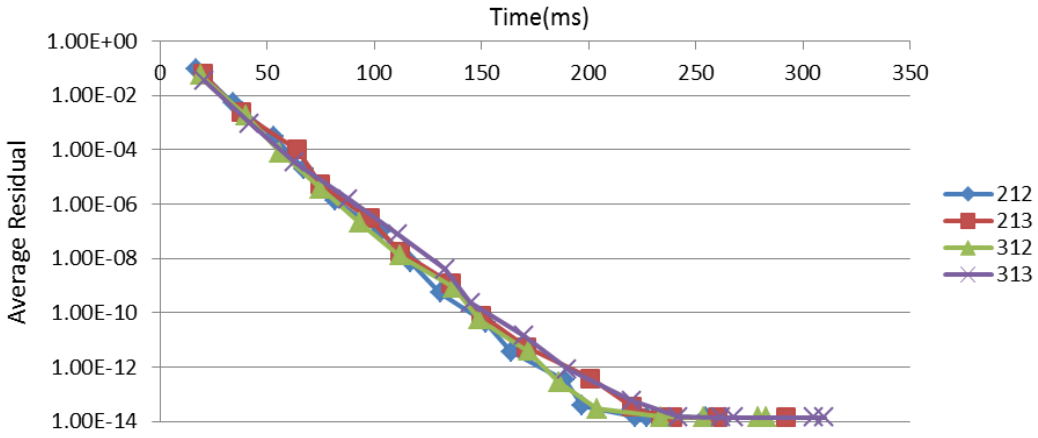

Figure 20: Effect of smoothing steps on the performance of the GMG method

In the following we analyze the effect of the floating precision on the performance of the GMG method (Figure 21). We considered single and double precision, in combination with the best performing smoothing configurations (212 and 312). The average residual in single precision is limited to approx. 1e-5, whereas in double precision it decreases to 1e-14. The residual of 1e-5 is reached slightly faster in single precision since the GTX Titan Black card has a higher GFLOP processing power in single precision than in double precision.

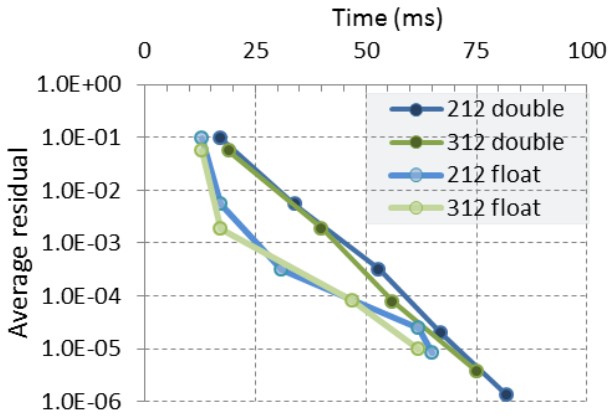

Figure 21: Effect of floating point precision on the performance of the GMG method

Another important aspect is the effect of the stencil configuration on the performance of the GMG method (Figure 22). We considered the 7-point stencil with RBGS smoother and the 19-point and 27-point stencil with Jacobi smoother. The 19-point and 27-point stencils require 40% and respectively 155% more execution time to reach a similar level of accuracy. However, opposed to the analysis performed for different GMG schemes, different number of smoothing steps, and single/double floating point precision, we cannot state that one approach performs better than the others. Practically, for each stencil configuration a different problem is solved: although the analytical equations are the same, the different discretization schemes lead to different numerical equations.

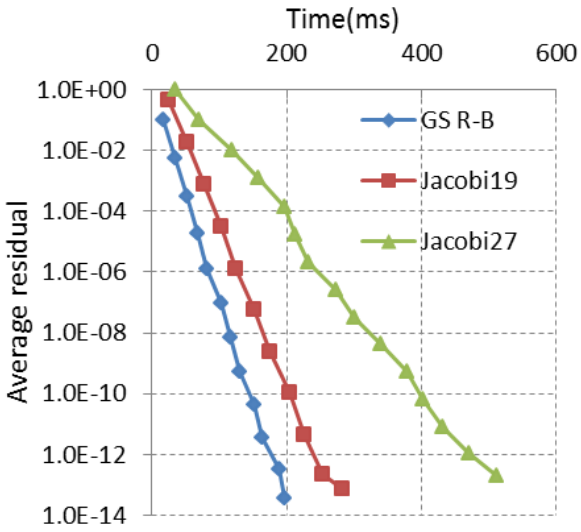

Figure 22: Effect of stencil configuration on GMG performance

Finally, we compare the performance of the best performing GMG variant (V-cycle, 312 smoothing steps) with the optimized PCG method, in double precision. We considered different fine grid resolutions and the three different discretization schemes. The results in Table 1 indicate that on the fine grid of 129x129x129 GMG offers a speed-up of 7.1x-9.2x over PCG, while it also leads to a smaller average residual. The speed-up is smaller on the intermediate grid, while on the coarse grid, PCG performs slightly better. This is given by the fact that for GMG the parallelism on the coarse grid (33x33x33) is limited, whereas for PCG the size of matrix A is still large enough to utilize the computational power of the GPU. In practice, typically used grids have more than 1 million nodes, case in which the GPU based GMG implementation performs better than the GPU based PCG implementation. While GMG offers these execution time advantages over PCG, it requires information regarding the underlying PDEs to be solved in order to generate different levels of discretization. On the contrary PCG only requires information regarding the discretization on the fineste level. Thus, the higher performance of GMG comes at the cost of a tighter link with the specific mathematical model which has to be solved numerically.

Table 1: Execution time and average error comparison for GMG and PCG

Fine grid resolution		129x129x129		65x65x65		33x33x33	
Method		PCG	GMG	PCG	GMG	PCG	GMG
RBGS	Avg. Error	5.22e-12	1.44e-14	1.41e-11	1.66e-14	4.21e-11	1.43e-14
7p stencil	Time [ms]	1118	121	124	50	28	33
Jacobi	Avg. Error	5.21e-12	7.00e-14	1.25e-11	6.99e-14	3.89e-11	6.79e-14
19p stencil	Time [ms]	1255	172	127	48	28	32
Jacobi	Avg. Error	4.30e-12	1.49e-13	1.40e-11	1.17e-13	2.94e-11	1.19e-13
27p stencil	Time [ms]	1502	211	145	61	29	35

8.2 GPU accelerated information retrieval using Bloom filters

8.2.1 Introduction

Semantic indexing is a popular technique used to access and organize large amounts of unstructured text data. Semantic search seeks to improve document retrieval accuracy by understanding searcher intent and the contextual meaning of terms as they appear in the searchable data space, whether on the web, or within a closed system.

The Bloom filter [Bloom, 1970], conceived by Burton H. Bloom in 1970, is a space-efficient probabilistic data structure (a binary array) that is used to test whether an element belongs to a set. Bloom Filter based algorithms are part of the major families of string matching with specific features, that makes them a great candidate for applications such as distributed databases or cache-membership protocols: in checking whether a data item exists at a specific location, one needs only to receive a small bit array to be checked locally, without unnecessarily querying the remote database. However, string matching technology now encounters new challenges because of increasing amounts of data and stringent response time requirements. This is the reason why improving the underlying implementation of string matching algorithms becomes critical.

Research activities into GPU accelerated string matching for applications in information retrieval discuss the utilization of programmable pipeline Graphics Processing Units (GPUs) for high speed string matching [Gee, 1987]. The process of mapping and searching is responsible for a large percentage of the computational load even though it is mathematically and algorithmically fairly simple. Therefore, increasing the speed of these operations could improve computing performance.

The goal of the current work was to evaluate the viability of using GPUs to speed-up the Bloom filter based string searching algorithm. Moreover, to increase the information retrieval accuracy, several preprocessing techniques have been considered.

8.2.2 Methods and implementation

The Bloom filter offers a compact probabilistic approach to represent a set that can result in false positives (claiming an element to be part of the set when it was not actually inserted), and no false negatives (i.e. it never reports an inserted element to be absent from the set). The probability of false positive results can be controlled.

The basic operations involve adding elements to the set and querying for element membership in the probabilistic set representation. The accuracy of a Bloom filter depends on the size of the filter, the number of hash functions used in the filter, and the number of elements added to the set [Broder et al., 2003].

Although the Bloom Filter may report false positive results, the false-positive probability can be controlled (lowered) by choosing proper filter parameter values.

The two main operations of the Bloom filter, the generation of the filter from a set of words, and the testing operation, can be parallelized. This inherent parallelism of the algorithm can be exploited on a GPU, which contains several streaming multiprocessors, each of them containing several cores. In the following we introduce the overall workflow of the CPU-GPU based hybrid Bloom filter implementation (Figure 23).

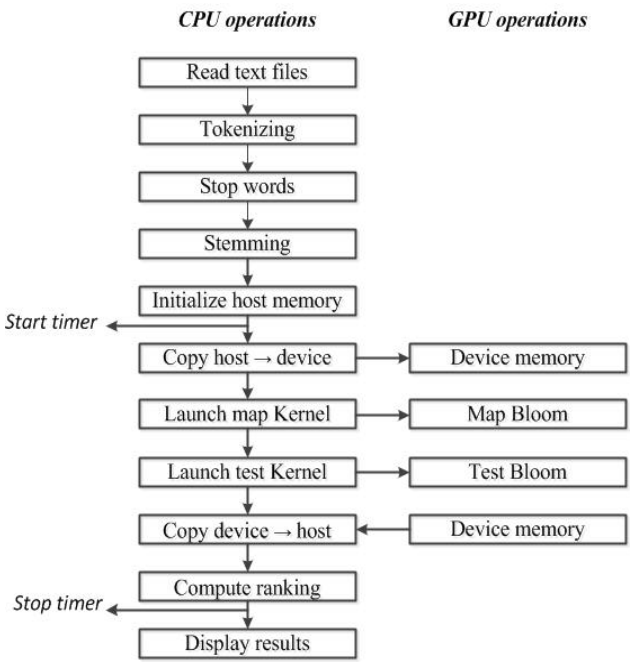

Figure 23: Workflow of the CPU-GPU version

First, several text files are read, which are subsequently used to generate associated Bloom filters. In the following several preprocessing techniques are applied. First, the text is tokenized: the stream of text is broken down into words, and spaces and punctuation marks are removed. Next, a predefined set of stop words [*Stop words, 2009] is removed from the input data. Finally, stemming is applied in order to reduce inflected or derived words to their word stem, base, or root form.

The threads in the execution configuration for the Bloom filter generation are organized as follows: each thread block processes one input file, and each thread processes one word. If a text file contains more words than threads, each thread will process several words. For a precise identification of the input data to be used by each thread of the kernel, aside from the array containing the text information, two additional arrays are precomputed and transferred to the GPU: an integer array containing the start index for each word in the char array, and a second integer array containing the start index for each file.

Once the Bloom filter has been generated, it is stored in the global memory, and another kernel is launched which performs the query operation. The execution configuration is set as follows: each thread block processes one Bloom filter vector, and each thread tests a query word for its membership.

Next, the query results are copied back to the host memory. Finally, the documents are ranked based on their relevance relative to the query, and they are displayed to the user. Relevance rankings of documents in a keyword search is calculated using the assumptions of document similarity theory [Turnley et al., 2010], by comparing the deviation of angles between each document associated vector and the original query vector, where the query is represented in the same vector space model.

In the following, we focus on the filter generation kernel (*Algorithm 1* displays the pseudocode of the map Bloom kernel function). In the mapping process, when scanning the corpus, each word is hashed against its corresponding Bloom filter vector stored in global memory by a separate CUDA thread (lines 6 to 9 in *Algorithm 2*). Because each word is mapped simultaneously by different threads, this leads to multiple accesses to the same memory buffer. To avoid race conditions, atomic operations [*NVIDIA, 2015] are used.

Algorithm 2. Parallel Bloom Filters mapping operation

1: Input: words from text files
2: Input: index of each word beginning
3: Input: index of each file beginning
3: Output: Bloom vector for each file
4: for all text files do
5: initialize all-zero *bitVector* of size *m* bits
6: for each *w*ord in current text file (denoted *x*) do
7: for each hash function *h* do
8: $bitVector[hash_h(x)] = 1$

```
 9:            end for
10:        end for
11:  end for
```

Since global memory operations have a large latency, and some of these operations are performed atomically, we also consider an optimized implementation, for which shared memory is employed. Instead of performing write operations on the filter in global memory, each thread block uses a local shared memory array to generate a local filter. This is possible since the global filter has a preallocated segment for each file. Once all threads of a thread block have finished the processing (a synchronization barrier is used), the locally generated Bloom filter is copied into the global memory, into the corresponding segment. Atomic operations are still required, but since they are performed in the shared memory, the latency is significantly lower.

8.2.3 Results

To evaluate the performance of the different implementation versions of the Bloom filter application, we used an Intel i7 8 core at 3.4 GHz, and a NVIDIA GTX Titan Black graphics card. To test the above-mentioned developments, a publicly available benchmark dataset consisting of unstructured text files were used, gathered from articles in various scientific domains such as bioinformatics, molecular physics, astronomy, etc. Experimental tests revealed that the average number of words per article is of around 330.

To process the text files, the map Bloom kernel generates a number of thread blocks equivalent to the number of mapped files, whereas each block of threads contains 512 threads. If the number of files is greater than the maximum number of thread blocks, the data is processed in sequential batches. Similarly, if the number of words in a document is larger than 512, each thread processes several words.

Next, we focus on the comparison of the CPU and the hybrid CPU-GPU implementation of the Bloom filter application. Computational results were perfectly identical for both versions. First, we analyze the execution time improvements for the parts that were ported to the GPU. All execution times reported below are computed as average value over 10 runs.

Table 2 displays the execution times results for the mapping operation, when different number of files are used. Three different versions were considered: the CPU version, the GPU version which uses only global memory, and the optimized GPU version which additionally employs shared memory. For the latter two, we additionally display the speed-up. One can observe that both GPU based implementations significantly outperform the CPU based implementation: execution time decreases by more than two orders of magnitude (the speed-up stabilizes at around 300x). Secondly, if more than 500 files are used the optimized GPU based version is roughly three times faster than the baseline GPU version. If the number of files is smaller, the differences become even larger. This performance gap is given by the latency of the global memory operations.

Table 2:Average execution time and speed-up analysis for the map bloom operation, for varying number of files

Number of files	CPU exec. time [ms]	Baseline GPU version		Optimized GPU version	
		Exec. time [ms]	Speed-up	Exec. time [ms]	Speed-up
50	66.9	3.793	0.288	17.638	232.292
100	100.3	4.891	0.476	20.507	210.714
250	263.3	6.041	1.051	43.585	250.523
500	579.3	7.708	1.905	75.156	304.094
750	893.7	9.389	2.773	95.186	322.286
1000	1162.1	11.120	3.681	104.505	315.702
2000	2390	18.144	7.050	131.724	339.007

Next, we analyzed the effect of the number of hash functions on the execution time of the mapping operation (the number of files is maintained constant at 1000). Since the optimized GPU based version outperforms the baseline implementation, in the following only this version is considered. The number of hash functions directly influences the false positivity rate, which is also displayed in Table 3. The results indicate that even when the number of hash functions decreases drastically, the speed-up is still maintained at a high level (around 200x). This is given by the fact that the hash functions are computed for a single word in a sequential manner, both on the CPU, and on the GPU.

Table 3: Average execution time and speed-up analysis for the map bloom operation, with varying number of hash functions

Number hash functions (k)	False-positivity rate (p)	CPU execution time [ms]	Optimized GPU execution time [ms]	Speed up
2	0.2	170.527	0.874	195.060

3	0.1	267.936	1.224	218.936
4	0.05	395.727	1.574	251.494
7	0.01	710.218	2.627	270.330
8	0.005	844.164	2.977	283.599
10	0.001	1089.273	3.681	295.938
11	0.0005	1198.200	4.039	296.657

The second kernel focuses on the search operation. Typically, the mapping process is performed offline, while the search is performed online. Hence, the execution time results are especially critical for the latter operation. Table 4 displays the execution time and speed-up results obtained for a search request containing 20 words, and for a Bloom filter based on varying number of files. The speed-up is significant, although not as large as for the mapping operation. Again, a relatively stable speed-up value, approx. 20x, is reached when over 500 files are used in the search operation.

Table 4: Average execution time and speed-up analysis for the query operation, with varying number of files

Number of files	CPU execution time [ms]	GPU execution time [ms]	Speed up
50	0.775	0.157	4.926
100	1.326	0.155	8.556
250	3.655	0.237	15.408
500	6.106	0.341	17.907
750	8.807	0.442	19.908
1000	11.024	0.554	19.897
2000	20.514	0.908	22.603

Next, we kept the number of files constant (at 1000) and varied the number of words in the query. The results are displayed in Table 5. An almost linear dependence between the speed-up and the number of words can be observed.

Table 5: Average execution time and speed-up analysis for the query operation, with varying number of query words

Number of query words	CPU execution time [ms]	GPU execution time [ms]	Speed up
5	3.102	0.480	6.465
10	5.875	0.522	11.244
15	8.731	0.529	16.493
20	11.024	0.554	19.897
25	15.097	0.565	26.716

Finally, we display in Table 6 the execution time of the entire application, consisting of both mapping and querying operations, in the configuration with 2000 files, 10 hash functions, and 20 words in the search query. One can observe that the reading and preprocessing of the files takes the majority of the execution time. However, since this step is performed offline, it is not critical. Next, for the map Bloom operation, for the GPU based workflow two execution times are displayed: one for copying the input data to the global memory of the GPU, and one for the actual mapping operation. Similarly, for the query operation, once the search results have been determined, the results are copied back to the CPU for ranking and visualization.

Table 6: Execution time for each operation

Operation	CPU execution time [ms]	GPU execution time [ms]
Read + preprocess files	11951.3	
mapBloom	2390	8.744 (copy)
		7.050 (exec.)
testBloom	32.9	0.908 (exec.)
		0.273 (copy)
Compute Ranking	9.6	
Total	14383.8	11977.904

8.3 GPU–Accelerated Texture Analysis Using Steerable Riesz Wavelets

8.3.1 Introduction

Textured pattern recognition is a key research topic in computer vision. One of the fundamental concepts in this area is the characterization of the local organization of image scales and directions to identify patterns [Blakemore et al., 1969]. Although much research has been carried out on this topic [Crick et al., 1980], [Zhang et al., 2007], it has proven difficult to elegantly exploit the potential of local attributes for classification. Depeursinge et al. proposed an iterative multi–scale and rotation–covariant texture learning approach using steerable Riesz wavelets [Depeursinge et al., 2014], [Depeursinge et al., 2013]. The framework allows learning local computational models (or distinctive signatures) of patterns. Subsequently, the models yield feature vectors that are optimally discriminant for a given problem. Classification accuracies of up to 98.4% were reported on databases such as Outex1. However, the algorithm requires days of computation for large databases. To improve its usefulness, the computations have been parallelized on GPUs.

8.3.2 Methods and implementation

Nth–order Riesz wavelets are used here to learn the texture signatures. This starts with the convolution of Riesz filters with training images in order to obtain Riesz coefficients. Their respective energies are derived and used to create Support Vector Machine (SVM) models, while following a one–versus–all (OVA) supervised learning strategy [Depeursinge et al., 2014]. Using the steerability property of Riesz wavelets, they are tuned to achieve local alignment at each pixel. The aligned coefficients form part of the final feature vector that can be used for texture classification.
Table 7 outlines the algorithm components and their percentage of the total run time. Alignment of signatures is the most computationally intensive operation of the application. Figure 24 illustrates the workflow for signature alignment. For each pixel, the computation of the maximum response requires solving a Nth–order trigonometric polynomial [Depeursinge et al., 2014]. Since the alignment can be performed independently for all pixels, the GPU becomes a well-suited option for greatly decreasing run time.

Table 7: Percentage of time spent by each major component within cpu based application

Component	%Time
Generation of Riesz energies	0.03
Creating matrices and computing SVM coefficients for ONE versus ALL (others) comparison	0.001
Alignment of signatures (see Equation 4)	99.289
Evaluation using SVM and post-processing	0.67

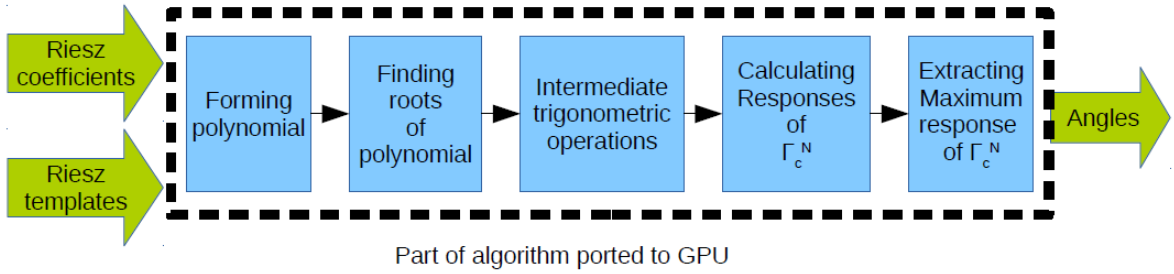

Figure 24: Part of the workflow around the pixel–wise local alignment of Riesz coefficients using the Riesz templates, Gamma

We first introduce a baseline GPU based implementation of the texture learning approach where all computations are performed in double precision. The GPU based implementation (called *GPUBase*) covers the actual computation of the feature vector. Since computing the responses takes around 99.28% of the execution time, it represents the main focus of the parallelization activities. A significant amount of data is required to reside on the GPU (requiring expensive copy operations), whereas the number of actual computations needed for computing the responses is small. This limits the throughput and we identify as main approach for performance improvement an increase in compute intensity [Zhong et al., 2014]. To increase compute intensity the implementation covers also the computation of polynomial roots and the maximum response extraction, along with a series of other operations that include trigonometric operations applied to the computed roots. The *GPUBase* version is divided into two parts: the computation of polynomial roots and the feature vector computation. In the

first part, each thread stores all N + 1 coefficients of the associated polynomial, and all real roots are computed. To determine these roots, we combine Newton's method that allows us to approximate one root of a polynomial with Horner's method for polynomial long division [Gautschi, 1997]. In the second stage, the feature vector is built by regional averaging of the energies. Each thread updates locations (based on the number of real roots found by the thread) from the feature vector. Next, the local orientations of each template are optimized to maximize their response, which is carried out by aligning Riesz components based on the dominant orientation of the signatures [Depeursinge et al., 2014]. For the *GPUBase* version these steps are included in the kernel, since no additional data are required.

Since data in the GPUBase version are stored in the global memory, and the performance of the kernel is primarily affected by the global memory bandwidth, we first address this aspect.

1) Register Usage: We use registers to store data that are otherwise repeatedly loaded: for the *GPUBase* version this strategy is applied for the values computed through trigonometric and associated operations. Since several intermediate computations are performed, by placing the partial results into registers, and merging the repetitive loop structures, a new enhanced implementation (*GPUReg*) is obtained (Figure 25). To further reduce global memory access, we introduce another version (*GPURegGlM*), in which we store the arc-tangent value for a root in an additional register. First the data are read from the register and stored into the global memory as it is required in the end, then, for the sine and cosine operations, the data cached in the register are used.

```
// global memory buffers
cosT[n * dimx * dimy];
sinT[n * dimx * dimy];
cossin[n*(n+1)* dimx * dimy];
template[n+1];
steermat[(n+1)*(n+1)*(n+1)];
orig[(n+1)* dimx * dimy];

void GPUBase(...)
{
...
R=roots(...);
for (int i=0; i<roots; i++)
  {
   tha[idx] = atan(R[idx]);
   cosT[idx] = cos(tha[idx]);
   sinT[idx] = sin(tha[idx]);
  }
for (int i=0; i<n+1; i++)
 for (int j=0; j<roots; j++)
    {
      cossin[index] =
       pow(cosT[idx],(n-i))*
       pow(sinT[idx],i);
    }
for(int i=0; i<n+1; i++)
 for(int j=0; j<n+1; j++)
  for(int k=0; k<n+1; k++)
   for(int l=0; l<roots; l++)
    {
     V[Vidx] += template[i]*
        steermat[steerIdx]*
        cossin[cossinIdx]*
        orig[origIdx];
    }
...
}
```

(a)

```
void GPUReg(...)
{
...
R=roots(...);
for (int i=0; i<roots; i++)
 {
  tha[idx] = atan(R[idx]);
  double cosT = cos(tha[idx]);
  double sinT = sin(tha[idx]);
 }
for (int j=0; j<n+1; j++)
  {
   double cossin =
     pow(cosT, (order-j))*
     pow(sinT, j);

   for (int k=0; k<n+1; k++)
    for (int l=0; l<n+1; l++)
      {
       V[Vidx] += template[i]*
          steermat[steerIdx]*
          cossin[cossinIdx]*
          orig[origIdx];
      }
  }
...
}
```

(b)

Figure 25: Simplified kernel code for (a) GPUBase, and (b) GPUReg

2) Shared Memory Usage: Since the storage of the majority of the data in shared memory could lead to exceeding the maximum amount of available memory when higher Riesz order and/or higher image resolution are employed, the utility of this type of memory is limited herein. The starting point for the new kernel is the *GPUReg* version. Only one data set (describing the weights of the Riesz components based on SVM coefficients) is small enough to not exceed the maximum size of the shared memory. The drawback of the first strategy, *GPUShM*, is that we write in the shared memory only if the local index is less than (N+1). Thus, if N is much lower than the total number of threads within a block only a small number of threads writes in the shared memory array, leading to divergent branches and hence to serialization. Shared memory can also be used to store data that cannot be put into registers and are repeatedly accessed by a thread [Vizitiu et al., 2014]. We observe that in GPUReg there are data (values within arc–tangent array) that are accessed multiple times by each thread. Because the values are not shared among multiple threads, we change the memory space in which data are stored without any

need of synchronization.

8.3.3 Results

We evaluate the different texture learning strategies using a hardware configuration based on an Intel Core i7 3.8 GHz processor with 64GB of memory and an NVIDIA GeForce GTX TITAN Black GPU, configured with 48KB of shared memory and 16KB of L1 cache, compute capability 3.5 and the CUDA toolkit version 6.0. The overall workflow of the application is implemented in Matlab, and the parallelizable components are implemented in C++ for the CPU based version, and in CUDA for the GPU-based versions. We first analyse the execution times for a configuration with a Riesz order of 8 and an image size of 128 x 128 (the CPU C++ based version, CPUBaseline, was considered alongside the 5 GPU based versions). We chose the total number of threads to be equal to image width x image height. Regarding the distribution of threads and blocks: while for most of the versions we adopted a standard number of 1024 threads per block, for the versions that use shared memory the number of threads is limited by the maximum size of this type of memory. The results are shown in Table 8.

Table 8: Execution times [s] of implementations for a single time step, when riesz order and image size are set to 8 and 128x128 respectively.

Method	Execution time [s]	Method speed–up
GPUBase	0.088 ± 0.009	56-68x
GPUReg	0.058 ± 0.006	85-103x
GPURegGlM	0.065 ± 0.005	78-89x
GPUShM	0.105 ± 0.010	47-56x
GPUShMTha	0.071 ± 0.012	65-91x
CPUBaseline	5.413 ± 0.055	

All GPU–based versions lead to a significant speed–up compared to the CPU–based version. The best performance is obtained for *GPUReg*, leading to a significant reduction of the execution time (98.9%), as compared to *CPUBaseline*. *GPUReg* improves data reuse and reduces global memory accesses by employing additional registers. The *GPURegGlM* version continues to reduce global memory load operations by using an additional register and as a result the execution time compared to the baseline *GPUBase* is decreased by 26%, but it increases slightly when compared to GPUReg: the larger number of registers limits the number of blocks of threads that can run simultaneously. Next, shared memory is used in two versions to reduce latencies and global bandwidth usage: the first version (*GPUShMTha*) is limited by the shared memory size and therefore occupancy decreases, while the second implementation (*GPUShM*) performs slower than the baseline GPUBase version due to the massive warp serialization requirements.

Next, we determined the computation time of the entire application, when using the best performing GPU based version *GPUReg* and the CPU based version *CPUBaseline*. The results are displayed in Table 9. The computation time decreases from 70.87 hours to 2.39 hours (speed–up of 29.58x). The Riesz order and the image size have a considerable impact on the execution time, since they affect the level of parallelism. We considered the best performing GPU based implementation and determined the speed–up of the parallelizable part for: (a) four different Riesz orders (8, 10, 12 and 14) with image size set constant at 128x128, and (b) four different image sizes (128 x 128, 256 x 256, 512 x 512, 1024 x 1024) with Riesz order N = 8 (Figure 26). When the Riesz order increases from 8 to 14 the speed–up changes from 93x to 148x. Similarly, as the image size increases to 1024 x 1024, the speed–up increases to 235x. The results indicate that once the image resolution increases beyond a certain threshold the speed–up curve flattens since the occupation of the GPU decreases substantially (due to the larger number of registers).

Table 9: Execution times [s] of the entire matlab riesz–based texture classification where the parallelized parts implemented in c/c++ and cuda were integrated

Component	Sub-component	Execution time [s] MATLAB + CPU	MATLAB + GPU
Generation of Riesz energies		78.884	78.884
Creating matrices and computing SVM coefficients		3.32	3.32
Alignment of signatures	Pre-processing	1574.06	1574.06
	Alignment of Riesz coefficients	249431.04	2897.058
	Post-processing	2340.09	1729.721
Evaluation using SVM		1729.721	1729.721
Total		255157.115	8623.133

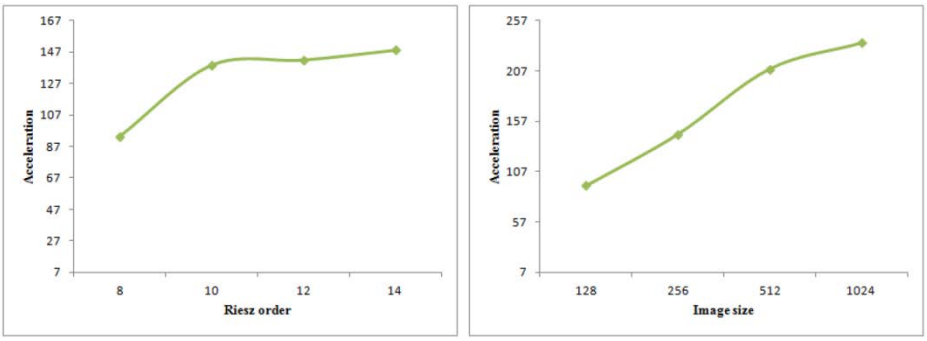

Figure 26: Comparison of speed–up obtained with the best GPU based implementation over the CPU based implementation with different (a) Riesz orders and (b) image dimensions

8.4 GPU-accelerated model for fast, three-dimensional fluid-structure interaction computations

8.4.1 Introduction

Fluid-Structure interaction (FSI) consists in simulating the fluid flow in a domain with moving boundaries. The boundary displacement may be known and directly enforced (one-way FSI) or may be computed from the interaction with a solid model which uses as input the stresses given by the fluid (two-way FSI). For arterial circulation simulations, the vessel wall is usually modelled as a viscoelastic material and the displacements are computed using a finite element method (FEM) based solver. The other approach consists in performing a one-way coupling, whereas the vessel wall displacement is directly extracted from patient-specific image data and is enforced onto the flow simulation. Similar work has been done in [Lantz et al., 2014] where a commercially available FEM based solver was used to simulate the blood flow in the human aorta by incorporating patient-specific wall motion.

Herein, we present an efficient workflow for embedding the wall motion, given as a set of polygonal meshes, into a Lattice-Boltzmann (LBM) simulation. We use a GPU accelerated LBM implementation for fast computations. We used the proposed method to perform simulations of the 3D peristaltic flow problem [Shapiro et al., 1969].

8.4.2 Methods and implementation

LBM is based on a discrete representation of the linearized Boltzmann equation on a regular Cartesian grid. The method consists of two steps, called collision (1) and streaming (2) which are applied at each grid point. The implementation is based on a multiple relaxation time (MRT) collision operator and a three-dimensional 19-velocity lattice [d'Humieres et al., 2002]. For more information on LBM we refer the reader to [Yu et al., 2003]. For the one-way FSI approach the wall motion is given as input data and is not influenced by the flow properties. Herein, the time-varying geometry is given as a set of polygonal meshes, each describing the wall position at a moment in time. Based on the given time samples, we compute the Discrete Fourier Transform (DFT) for each node in the mesh. The wall position and velocity can then be determined at any given time by evaluating the inverse Fourier transform. The motion of the wall is usually both periodic and smooth, hence the Fourier spectrum of the given time-varying geometry contains a small number of modes. This makes the DFT based approach more

convenient than a regular interpolation between the time samples. The DFT is only computed in the pre-processing stage, whereas during the simulation only the inverse transform is evaluated to reconstruct the geometry at the current time. To determine the wall velocity at each node, we compute analytically the time derivative of the inverse transform. The mesh is embedded in the Cartesian grid by computing the signed distance f(x) for each point x in the grid. This step is also exclusively done in the pre-processing stage. Lattice nodes are labeled as fluid or solid nodes and herein we assume that the distance f(x) < 0 for fluid nodes, and f(x) > 0 for solid nodes. The size of the grid is given by the spatial resolution dx and the size of the smallest bounding-box large enough to fit the geometry at any time. Given the signed distance at each grid node, the fluid region of the domain can be accurately identified. Depending on how complex the geometry is, the fluid region will regularly occupy only a small portion of the entire domain. To reduce memory requirements, we use a sparse grid implementation based on an indirect addressing scheme [Nita et al., 2013]. When the geometry is updated, the fluid region of the domain changes: some fluid nodes become solid nodes and vice versa. To avoid the requirement of regenerating the grid when the geometry is updated, all nodes that are labeled as fluid nodes for at least one-time step are identified in the pre-processing stage (all these nodes are considered when generating the sparse grid). As the mesh is changing in time, the grid nodes need to be updated as well. This process consists in looping over all facets and computing f(x) for each grid node located close to the current facet. This operation updates the distance function only for a few layers of nodes close to the boundary, while the other grid nodes are not affected. It is not required to update all grid nodes since the exact distance to the boundary is only required at the boundary nodes i.e. nodes that have a neighboring node located outside the mesh.
Since updating the geometry is a computationally expensive operation, it is not performed at each time step. The time interval after which the geometry is updated is chosen so as to have a maximum displacement smaller than 0.5dx. This significantly improves performance since the time-step of one LBM iteration is much smaller than the time interval for updating the geometry. Since the wall velocity is described by its Fourier spectrum, it is not straightforward to find the maximum velocity analytically. Instead we used a numerical approach that computes the value iteratively. When the geometry is updated, the wall velocity u_w is associated to each grid node close to the wall, along with the new signed distance f(x). The wall velocities are enforced on the fluid boundary nodes during the streaming step.
All computational steps are implemented with CUDA and run on the GPU.

8.4.3 Results

We applied the model to study 3D peristaltic flow: a configuration in which a periodic deformation of the walls generates net fluid motion. Here, the geometry is given by a spatially periodic cylindrical vessel, where a periodic deformation function is applied to its walls. The experiment setup was previously described in [Connington et al., 2009]. In Figure 27, we present the geometry along with the flow vectors. Both the inlet and outlet were periodic boundaries. The fluid motion is given only by the wall deformation.
Figure 26 displays the results: the simulated flow rates match the known solution closely, with a maximum absolute error of 0.17 mm3/s, and an average absolute error of 0.11 mm3/s. Furthermore, we performed experiments with different values for the Reynolds number, to observe its effect on the flow regime. The Reynolds number was changed by adapting the wave speed, while the other parameters were maintained constant.

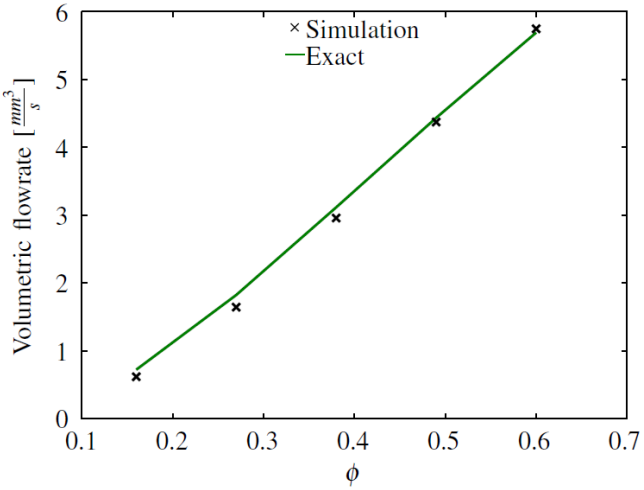

Figure 27: Peristaltic flow simulation. Comparison between the measured average flow rates and the exact solution

In Figure 28, we present the streamline plots of the flow with *Re* varying between 1 and 100. To observe the closed streamlines, the flow is viewed in a frame of reference moving at half of the wave speed. The two counter-rotating vortices decrease in size as the Reynolds number increases. Furthermore, the pumping efficiency was found to decrease. When compared to the analytically computed flow rate, the measured flow rate was 13% and 35% smaller for Re = 10 and Re = 100 respectively. The closed streamlines were previously reported in [Connington et al., 2009] but under different circumstances, characterized by high amplitude ratio, where the fluid is trapped inside the bolus and is moving along with the wave. The simulations were run on a commercially available graphical processing unit (GeForce GTX TITAN Black) with a spatial resolution of dx = 0.15mm and a time step of dt = 57ms. The average execution time was of around one hour for one simulation, which consisted of five wave periods. Roughly 50% of the total execution time accounts for the geometry update process.

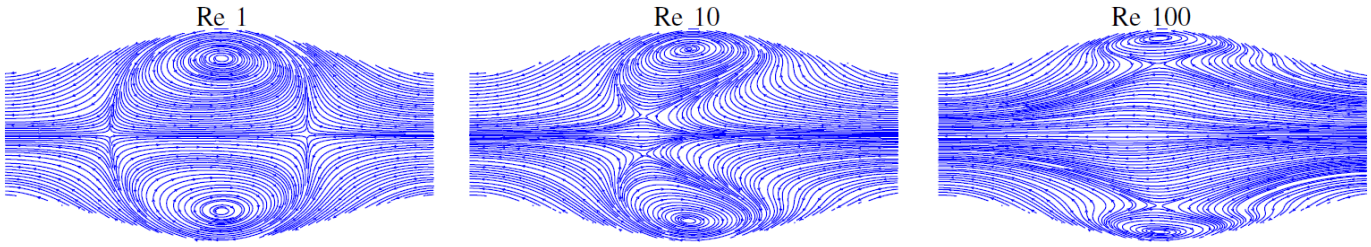

Figure 28: Peristaltic flow streamlines for different Reynolds numbers when the reference frame is moving at half of the wave speed

8.5 GPU-accelerated voxelizer

8.5.1 Introduction

Performing a Fluid-Structure interaction (FSI) using the Lattice-Boltzmann method (LBM) requires the moving geometry to be embedded into a Cartesian grid of uniformly distributed points using a signed distance field $\phi(\mathbf{x})$. However, the geometry is typically given as a sequence of non-uniform polygonal meshes. A surface voxelization operation is required to compute the distance field. The main challenge of voxelization consists in associating each vertex $\mathbf{v}_i$ of a polygonal mesh to each node $\mathbf{x}_i$ of the Cartesian grid. Typically, the size of the grid is between 500,000 and 50,000,000 nodes while the size of the mesh is between 50,000 and 300,000. This makes the voxelization a computationally expensive operation.
For the FSI computations, since the surface is moving, the voxelization operation is required at each solver iteration to update the position of the surface. With the classical method (CPU based implementation), the surface voxelization operation is the performance bottleneck as it occupies around 50% of the total computation time [Nita et al., 2015]. Therefore, it is crucial that an efficient implementation is developed.

8.5.2 Methods and implementation

8.5.2.1 The classical method

The Cartesian grid is defined from a three-dimensional image by its dimensions (N_x, N_y, N_z), an origin $\mathbf{o}$ and a grid spacing δx (the grid nodes are uniformly distributed hence $\delta x = \delta y = \delta z$). The grid size is chosen to satisfy the flow solver stability constraints. The origin $\mathbf{o}$ and the grid spacing δx is used to transform from physical coordinates to grid coordinates (and vice versa) i.e. to find the voxel (i, j, k) that corresponds to a point $\mathbf{p} = (p_x, p_y, p_z)$. The transformation is defined as follows:

$$i = \left\lfloor \frac{p_x}{\delta x} - o_x \right\rfloor, j = \left\lfloor \frac{p_y}{\delta x} - o_y \right\rfloor, j = \left\lfloor \frac{p_z}{\delta x} - o_z \right\rfloor, \quad (1)$$

Where $\lfloor x \rfloor$denotes the floor function. And the inverse transformation:

$$p_x = i\delta x + o_x, p_y = j\delta x + o_y, p_z = k\delta x + o_z, \quad (2)$$

A mesh is defined as a set of triangles $T = (\mathbf{v}_1, \mathbf{v}_2, \mathbf{v}_3)$, for each triange we compute an axis-aligned bounding-box (AABB) as $AABB = (\mathbf{v}_{min}, \mathbf{v}_{max})$ so that the triangle will completely fit inside it, furthermore the AABB is enlarged in all directions using a small value $(2 - 3\delta x)$ so that the triangle vertices will never be located exactly on the AABB wall.
The classic method for surface voxelization consists in simply looping over each grid node $\mathbf{x}_i$ in each AABB and computing the signed distance $\phi(\mathbf{x}_i)$. To find all the grid nodes inside the AABB, one needs to transform $(\mathbf{v}_{min}, \mathbf{v}_{max})$ to grid coordinates to get $(i_{min}, j_{min}, k_{min})$ and $(i_{max}, j_{max}, k_{max})$ and then loop over all i, j and k values located inside the bounds.
The signed distance function is defined as follows:

$$\phi(\mathbf{x}) = d(\mathbf{x}, \mathbf{x}_\perp) sgn[(\mathbf{x} - \mathbf{x}_\perp) \cdot \mathbf{n}], \quad (3)$$

Where $\mathbf{n}$ is the triangle normal and $\mathbf{x}_\perp$ is the closest point to $\mathbf{x}$ on the triangle. The second factor in the above expression represents the sign, i.e. it will be negative or positive depending on which side of the triangle, the point $\mathbf{x}$ is located. For adjacent triangles the AABBs will intersect and will result in multiple ϕ values for the same grid point $\mathbf{x}$, one value for each AABB that point $\mathbf{x}$ is included in (Figure 29). In this case the absolute minimum value of ϕ will be chosen.

For the GPU implementation, the loop that processes the mesh triangles is parallelized so that one GPU thread will process one triangle. However there are several downsides that causes very poor GPU utilization in this case. The main problem arises at the adjacent triangles where the AABBs intersect. In the intersection regions, there will be multiple threads that need to update the ϕ value at the same grid node $\mathbf{x}$. In this case a synchronization operation is required to ensure that only one thread will update one location at the same time. The synchronization operation drastically reduces parallelism and GPU performance.

The other limitation is given by the fact that each GPU thread will process a different number of grid nodes because of the different AABB sizes. More specifically, the number of the grid nodes in an AABB is influenced by the size and orientation of the corresponding triangle. To achieve maximum performance with a GPU based implementation, all the threads should execute the same operations.

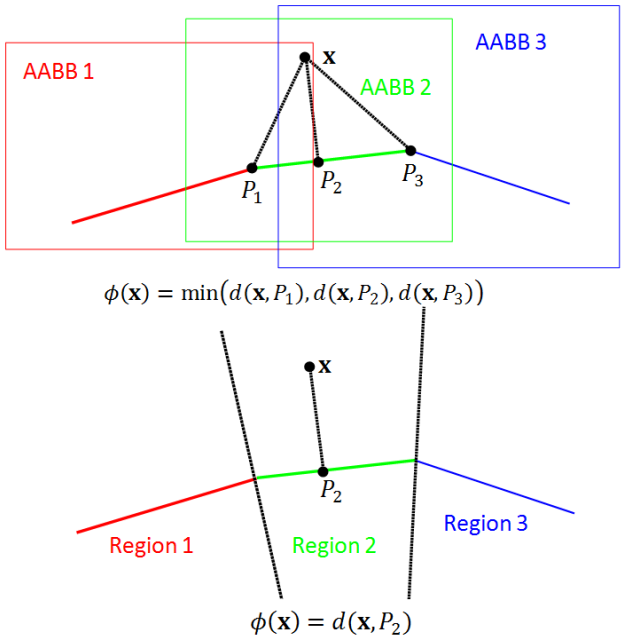

Figure 29: Two-dimensional analogy of the surface voxelization algorithm. The classic approach (up): φ is computed for all the nodes inside an AABB. And the separating planes technique (down): nodes that correspond exclusively to a facet are identified using sepa

8.5.2.2 The separating plane technique

The classical method can be improved by redefining the way grid nodes are associated with mesh triangles. Instead of computing the ϕ value for all the nodes in an AABB it is possible to identify a priori the nodes for which each mesh triangle will give the minimum ϕ. Hence, there will no longer be threads that will need to update ϕ at the same location $\mathbf{x}$. This method was initially presented in [Janßen et al., 2014].

For each triangle we define a region so that each point $\mathbf{x}$ in that region has the closest point $\mathbf{x}_\perp$ located on that triangle. To define such a region for a triangle, three planes are required, one for each edge. More specifically, if a node is located on the negative side of all three planes then that node is considered to belong exclusively to that triangle.

We check if a point $\mathbf{x}$ is located in a triangle region in the following way (Figure 30):

1. For each vertex $\mathbf{v}_i$ on the mesh, the vertex normal is computed as an angle weighted average of the normals of adjacent triangles:
   $\mathbf{n}_\Sigma = \frac{\sum \alpha_i \mathbf{n}_i}{|\sum \alpha_i \mathbf{n}_i|}$ (4)
2. For each edge $(\mathbf{v}_i, \mathbf{v}_j)$, with the associated vertex normals $(\mathbf{n}_i, \mathbf{n}_j)$ a separating plane is defined:
   $\mathbf{n}_s \cdot (\mathbf{x} - \mathbf{v}_i) = 0$ (5)
   Where $\mathbf{n}_s$ is the separating plane normal and is computed as an edge bi-normal:
   $\mathbf{n}_s = \frac{1}{2}[(\mathbf{v}_j - \mathbf{v}_i) \times (\mathbf{n}_i + \mathbf{n}_j)]$. (6)

3. A point $\mathbf{x}$ is considered to be located inside a region of a triangle $T = (\mathbf{v}_1, \mathbf{v}_2, \mathbf{v}_3)$ if it is located on the negative side of all three separating planes:

$$\begin{cases} \mathbf{n}_{s_1} \cdot (\mathbf{x} - \mathbf{v}_1) \leq 0 \\ \mathbf{n}_{s_2} \cdot (\mathbf{x} - \mathbf{v}_2) \leq 0. \\ \mathbf{n}_{s_3} \cdot (\mathbf{x} - \mathbf{v}_3) \leq 0 \end{cases} \quad (7)$$

For any two adjacent, non-intersecting triangles, the regions defined by (7) will not intersect. If each GPU thread processes the nodes in separated regions then there will never be any concurrency hence the synchronization is no longer required. This drastically improves the GPU parallelism and performance.

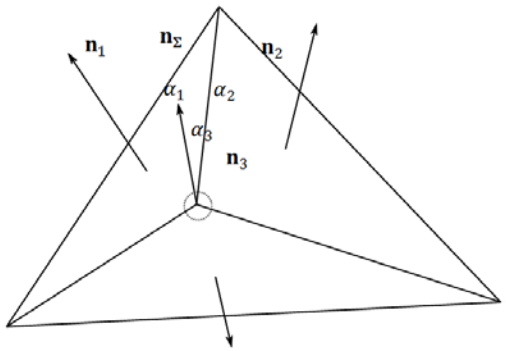

Figure 30: Defining a vertex normal as an angle weighted average of the normals from adjacent triangles

Fig. 1.5.2..

8.5.3 Results

To test our implementation we considered a known CFD benchmark case consisting of a large brain aneurysm [Steinman et al., 2013]. Figure 31 displays the mesh along with the voxelized surface.

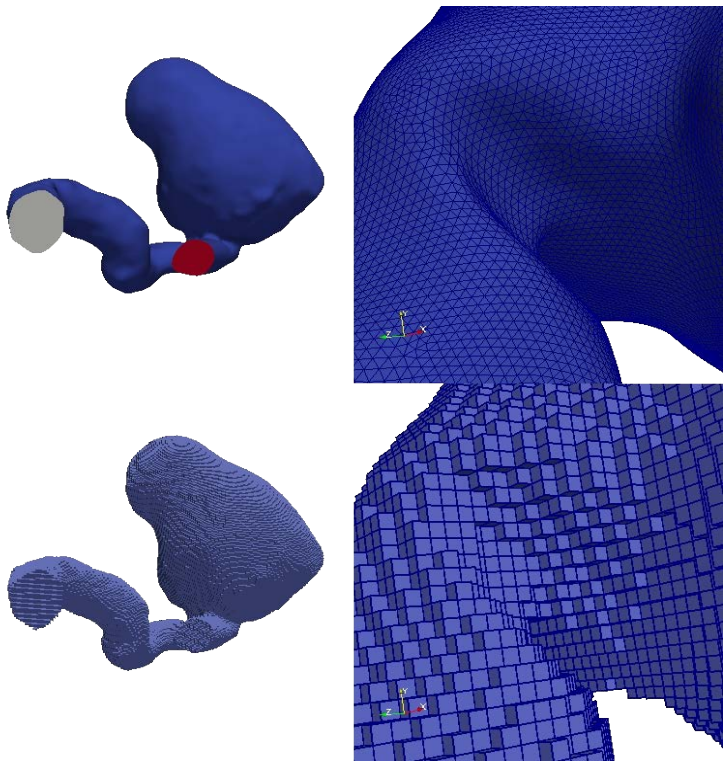

Figure 31: Test case: a large brain aneurysm mesh of 318,000 triangular elements (up) and the voxelized surface (down)

The mesh contains 318,000 triangular elements and the size of the grid in which the surface is embedded is 171x180x142. We performed the computations for this case using the CPU and GPU implementations for both the classic and the separating planes method. The hardware we used consists of an Intel i7 (8-cores) CPU and a GTX Titan Black GPU.
The execution times were:

- for the classic method on the CPU the execution time was 23.5 seconds and on the GPU it was 234 milliseconds which gives a speedup of around 100 times. The GPU execution time does not contain the CPU-GPU memory copy as in an FSI simulation the memory copy should only be done once in the pre-processing stage.
- for the separating planes method, the GPU execution time was 21.4 milliseconds. Compared to the current implementation that we use for FSI computations, the new GPU-accelerated one is around 1000 times faster. Using this approach, the performance of the FSI computations can be taken to an unprecedented level.

8.6 Publications

The methods, algorithms and implementations described above have lead to the publication of the following papers:

1. Vizitiu, A., Itu, L., Joyseeree, R., Depeursinge, A., Muller, H., Suciu, C. “GPU–Accelerated Texture Analysis Using Steerable Riesz Wavelets”, 24th Euromirco International Conference on Parallel, Distributed, and Network-Based Processing, Heraklion Crete, Greece, 2016.
2. Iacob, A., Itu, L., Sasu, L., Moldoveanu, F., Suciu, C. “GPU Accelerated Information Retrieval using Bloom Filters”, 19th Inter. Conf. on System Theory, Control and Computing - ICSTCC 2014, Sinaia, Romania, October 14-16, 2015, pp. 872-876.
3. Stroia, I., Itu, L., Niţă, C, Lazăr, L., Suciu, C. “GPU Accelerated Geometric Multigrid Method: Performance Comparison on Different Architectures”, 19th Inter. Conf. on System Theory, Control and Computing - ICSTCC 2014, Sinaia, Romania, October 14-16, 2015, pp. 175-179.
4. Stroia, I., Itu, L., Niţă, C, Lazăr, L., Suciu, C. GPU Accelerated Geometric Multigrid Method: Comparison with Preconditioned Conjugate Gradient, 19th IEEE High Performance Extreme Computing Conference, Waltham, MA, USA, Sept. 15-17, 2015, pp. 1-6.
5. Nita, C., Itu, L. M., Mihalef, V., Sharma, P., Rapaka, S., GPU-accelerated model for fast, three-dimensional fluid-structure interaction computations, Proc. of the 37th Annual Inter. Conf. of the IEEE Engineering in Medicine & Biology Society - EMBC 2015, Milano, August 25-29, 2015, pp. 965-968.

8.7 References

[Ament et al., 2010] M. Ament, G. Knittel, D. Weiskopf, and W. Strasser, “A parallel preconditioned conjugate gradient solver for the poisson problem on a multi-gpu platform,” in Parallel, Distributed and Network-Based Processing (PDP), 2010 18th Euromicro International Conference on, pp. 583-592, IEEE, 2010.
[Blakemore et al., 1969] C. Blakemore and F. W. Campbell, “On the existence of neurones in the human visual system selectively sensitive to the orientation and size of retinal images,” Journal of Physiology, vol. 203, no. 1, pp. 237–260, 1969.
[Bloom, 1970] B. H. Bloom, “Space/time trade-offs in hash coding with allowable errors,” Commun. ACM, vol. 13, no. 7, pp. 422–426, 1970.
[Briggs et al., 2000] W. L. Briggs, S. F. McCormick, et al., A multigrid tutorial. Siam, 2000.
[Broder et al., 2003] A. Broder, M. Mitzenmacher, “Network Applications of Bloom Filters: A Survey”, Internet Math., vol. 1, no 4, pp. 485-509, 2003.
[Chung, 2010] T. Chung, Computational fluid dynamics. Cambridge university press, 2010.
[Connington et al., 2009] Kevin Connington, Qinjun Kang, Hari Viswanathan, Amr Abdel- Fattah, and Shiyi Chen, “Peristaltic particle transport using the lattice boltzmann method,” Physics of Fluids (1994-present), vol. 21, no. 5, pp. 053301, 2009.
[Crick et al., 1980] F. H. C. Crick, D. C. Marr, and T. Poggio, “An information processing approach to understanding the visual cortex,” tech. rep., Massachusetts Institute of Technology, 1980.
[d’Humieres et al., 2002] Dominique d’Humieres, “Multiple–relaxation–time lattice boltzmann models in three dimensions,” Philosophical Transactions of the Royal Society of London. Series A: Mathematical, Physical and Engineering Sciences, vol. 360, no. 1792, pp. 437–451, 2002.
[Depeursinge et al., 2013] A. Depeursinge, A. Foncubierta-Rodriguez, H. Muller, and D. V. D. Ville, “Rotation–covariant visual concept detection using steerable riesz wavelets and bags of visual words,” in SPIE Wavelets and Sparsity XV, vol. 8858, 2013.
[Depeursinge et al., 2014] A. Depeursinge, A. Foncubierta-Rodriguez, D. V. D. Ville, and H. Muller, “Rotation–covariant texture learning using steerable riesz wavelets,” IEEE Transactions on Image Processing, vol. 23, no. 2, pp. 898–908, 2014.
[Gautschi,1997] W. Gautschi, Numerical analysis. Springer, 1997.
[Gee, 1987] A. Gee, “Research into GPU accelerated pattern matching for applications in computer security”, Univ. of Canterbury, Christchurch, New Zealand, 1987.
[Gui et al., 2012] Y. Gui and G. Zhang, “An improved implementation of preconditioned conjugate gradient method on gpu,” Journal of software, vol. 7, no. 12, pp. 2695-2702, 2012.
[Janßen et al., 2014] C. F. Janßen, N. Koliha, and T. Rung. A fast and rigorously parallel surface voxelization technique for GPGPU-accelerated CFD simulations. Communications in Computational Physics (accepted for publication), 2014.
[Lantz et al., 2014] Jonas Lantz, Petter Dyverfeldt, and Tino Ebbers, “Improving blood flow simulations by incorporating measured subject-specific wall motion,” Cardiovascular engineering and technology, vol. 5, no. 3, pp. 261–269, 2014.
[Nita et al., 2013] Cosmin Nita, Lucian Mihai Itu, and Constantin Suciu, “Gpu accelerated blood flow computation using the lattice boltzmann method,” in High Performance Extreme Computing Conference (HPEC), 2013 IEEE. IEEE, 2013, pp. 1–6.

[Nita et al., 2015] Nita, Cosmin, et al. "GPU-accelerated model for fast, three-dimensional fluid-structure interaction computations." Engineering in Medicine and Biology Society (EMBC), 2015 37th Annual International Conference of the IEEE. IEEE, 2015.
[Ruge et al., 1987] J. Ruge and K. Stüben, “Algebraic multigrid,” Multigrid methods, vol. 3, pp. 73-130, 1987.
[Shapiro et al., 1969] Ascher H Shapiro, Michel Y Jaffrin, and Steven L Weinberg, “Peristaltic pumping with long wavelengths at low reynolds number,” Journal of Fluid Mechanics, vol. 37, no. 04, pp. 799–825, 1969.
[Steinman et al., 2013] Steinman, David A., et al. "Variability of computational fluid dynamics solutions for pressure and flow in a giant aneurysm: the ASME 2012 Summer Bioengineering Conference CFD Challenge." Journal of biomechanical engineering 135.2 (2013): 021016.
[Trottenberg et al., 2000] U. Trottenberg, C. W. Oosterlee, and A. Schuller, Multigrid. Academic press, 2000.
[Turnley et al., 2010] P. D. Turnley, P. Pantel, “From Frequency to Meaning: Vector Space Models of Semantics”, Journal of artificial intelligence research, vol. 37, pp. 141–188, January 2010.
[Vizitiu et al., 2014] A. Vizitiu, L. M. Itu, C. Nita, C. Suciu, “Optimized Three-Dimensional Stencil Computation on Fermi and Kepler GPUs”, 18th IEEE High Performance Extreme Computing Conference, Waltham, MA, USA, Sept. 9-11, 2014, 978-1-4799-6232-7
[Yu et al., 2003] Dazhi Yu, Renwei Mei, Li-Shi Luo, and Wei Shyy, “Viscous flow computations with the method of lattice boltzmann equation,” Progress in Aerospace Sciences, vol. 39, no. 5, pp. 329–367, 2003.
[Zhang et al., 2007] J. Zhang, M. Marszalek, S. Lazebnik, and C. Schmid, “Local features and kernels for classification of texture and object categories: A comprehensive study,” International Journal of Computer Vision, vol. 73, no. 2, pp. 213–238, 2007.
[Zhong et al., 2014] J. Zhong and B. He, “Kernelet: High-throughput gpu kernel executions with dynamic slicing and scheduling,” IEEE Transactions on Parallel and Distributed Systems, vol. 25, no. 6, pp. 1522–1523, 2014.
[*NVIDIA, 2015] NVIDIA Corporation. “CUDA, Compute Unified Device Architecture Programming guide v7”, 2015.
[*Stop words, 2009] “List of English Stop Words”, 2009. [Online]. Available: http://xpo6.com/list-of-english-stop-words/

9 Conclusion

The Infostructure is currently in Beta version. Almost all the functionalities are in place and the last sprint will mainly concern tests and fixes of all these applications, as well as finalising the integration between them. This makes the infostruture status to “corresponding to the planning” despite the issues encountered during the whole project that could have slowed down the advancement. On a hardware standpoint, the current installation can provide all the validation and the addition of the last nodes will provide a ready to use base for the following.