A GPU based implementation of the spatial model checker VoxLogicA

Supervisor: Prof. Fabio Gadducci  
Supervisor: Dr. Vincenzo Ciancia  
Candidate: Laura Bussi

A.Y. 2019/20
Abstract

Recent years witnessed the widespread use of GPUs for speeding up computationally intensive tasks. As far as system verification is concerned, one of such tasks is model checking, a fully automated method to check the truth value of a logic formula at a given model. Some model checkers running entirely in GPU have been proposed, achieving a good performance improvement with respect to classical implementations. The thesis focus on the spatial model checker VoxLogicA, which has been successfully used to analyse real-world datasets of MRI scans of the human brain. We present an implementation of VoxLogica in GPU: it exploits existing algorithms for the labelling of connected components, which can be used to compute reachability-based spatial logic formulas on a graph.
# Contents

1 Introduction ................................................................. 3  
   1.1 Logics for image analysis ........................................... 3  
   1.2 Model checking on GPUs ............................................ 4  
   1.3 Thesis structure .................................................... 5  

2 Spatial Logics for Closure Spaces .................................... 6  
   2.1 About spatial logics .................................................. 6  
   2.2 Closure Spaces ...................................................... 7  
   2.3 Quasi-discrete closure spaces ..................................... 8  
   2.4 Syntax and Semantics of SLCS ..................................... 9  

3 VoxLogicA ................................................................. 12  
   3.1 Project structure .................................................... 12  
   3.2 Image Query Language ............................................... 15  

4 GPU based implementation .............................................. 17  
   4.1 GPGPU programming ................................................ 17  
   4.2 GPU architecture ................................................... 17  
   4.3 About the comparison between GPUs and CPUs .............. 19  
   4.4 VoxLogicA GPU based implementation ......................... 20  

5 Test results .................................................................... 29  
   5.1 A case study: medical image analysis ............................ 30  
   5.2 Image segmentation .................................................. 31  
   5.3 The BRATS benchmark ............................................... 31  

6 Conclusions .................................................................... 35
1 Introduction

1.1 Logics for image analysis

In the research field of Formal Methods in Computer Science, a large amount of work has been devoted to find an automated way to reason about reliability of hardware and software systems [1]. As the computational power of devices increases, these systems grow in complexity and increasingly involve critical tasks, whose correctness becomes then crucial: let us think, for instance, about software monitoring air traffic or software implementing security protocols. It is therefore important to provide automated and efficient tools which can be used to formally check whether a system satisfies a given property.

Model checkers are a fully automated tool which can be used to this aim [2]. In model checkers, we are provided with a model of a system and a logic formula expressing the property of interest, often stating that some event may/should happen at a certain moment in time. Consider, for instance, a mutual exclusion algorithm: a crucial property that must hold is the fact that two threads never write the same variable at the same time, i.e. at each moment in time at most one thread can modify the variable.

Such properties can be modeled by using temporal logics. The logics include boolean operators, together with other ones (called modal operators), allowing to reason about temporal properties. Consider the previous mutual exclusion example: as in temporal logics we are provided with an always operator, one may express the mutual exclusion property as “always, if a thread is writing the variable, no other thread writes it”. Several tools performing model checking, using different approaches, have been proposed, as in [3, 4].

In recent times a different kind of logics, dubbed spatial logics, received the attention of the formal methods community [5, 6]. It is meant to express properties about the topology of a system specified as a graph. Here, modal operators mirror in some sense the ones of temporal logics, but they are given a different semantics in order to express properties about space, for instance, the possibility to reach some point of the graph. A model checker for the verification of both spatial and temporal properties have been proposed in [7].

VoxLogicA [8] is a model checker for analysing images using spatial logics. An image is seen as a graph: each pixel (voxel, in the 3D case) is seen as a node connected to all of its neighbours. Logic formulas are then used to express properties of pixels (for instance, the fact that a black pixel is
surrounded by white ones). Spatial operators are provided to the user via a declarative language, called ImgQL (Image Query Language). ImgQL also provides operators for manipulating images, for instance to select a certain component or to apply a filter. Therefore, complex tasks can be managed by arbitrarily nesting such operators.

1.2 Model checking on GPUs

GPUs are highly specialized processing units. A GPU can easily have several hundreds of cores, each one performing the same operation on a different portion of data, hence adopting the SIMD (Single Instruction Multiple Data) model. GPUs then suit well data parallel task, for instance rendering a scene on a computer screen, but their performances can quickly degrade as cores compute more complex programs, involving branching and random accesses to memory.

General-Purpose computing on Graphics Processing Units (GPGPU) has gained popularity in recent years. For example, neural networks training has known a good improvement with this kind of devices. Nowadays, many ML libraries, as e.g. TensorFlow [9], Theano [10] or PyTorch [11] provide (transparent) GPU support to the programmer, in order to improve neural networks training performances.

Recently, GPUs began to be exploited in model checking too. Due to GPUs efficiency in performing linear algebra operations, efforts have been spent in implementing probabilistic model checking on these devices since 2010 [12, 13]. More difficulties arise when dealing with classical model checking: in this case, GPU implementations require to develop new algorithms and data structures, as done in [14, 15]. The proposed tools outperform CPU based model checkers, thus showing GPUs potentialities for this kind of tasks.

In general, the usage of GPUs could lead to achieve better performances for tools based on formal methods, as long as the considered tasks show a good degree of parallelism. This is the case of VoxLogicA, as its main task is to manipulate images and check properties over them: indeed, image manipulation is one of the tasks which GPUs are devoted to, so it is fair enough to expect an improvement in VoxLogicA performances by switching to this kind of devices. Of course, this means to rework the code taking into account the peculiarities of such devices.

Obviously, moving to a GPU based implementation carries several difficulties. The execution model does not allow to make assumption about the state of a thread’s neighbourhood, thus making necessary to use particular
synchronization mechanisms in order to guarantee the correct execution of the code. Furthermore, data movements between host and device must be handled carefully, in order to avoid unnecessary ones which could quickly degrade performances. Also, as VoxLogicA is an object oriented dotnet project, it is necessary to define a way to conveniently handle GPU calls and memory management in its environment.

1.3 Thesis structure

In section 2 we introduce the theoretical background of VoxLogicA, presenting some basic notions about topological spaces and spatial logics. In section 3 we explain the structure of the standard implementation of VoxLogicA, which is used as a basis for the GPU implementation. In section 4 we recap some notions about GPUs and GPU programming. We then present the GPU implementation of VoxLogica with some experimental results (section 5) as the original contribution of this thesis.
2 Spatial Logics for Closure Spaces

Spatial Logics for Closure Spaces has been introduced by Ciancia et al. in [6], in order to formally verify spatial properties of points and regions of discrete spatial structures, such as images or graphs, exhibiting physical properties, e.g. distance, proximity or reachability. In order to do that, the authors follow the modal logics approach, where some special operators are used to express properties about the neighbourhood of a certain point. Modal logics are often intended to express properties about time; in order to take into account a physical space, SLCS relies on the concept of Closure Spaces, which allows to uniformly treat topological spaces and graphs.

Along with this feature, the authors introduce a new operator, called Surrounded, specifying the fact that we cannot “escape” from an area of points satisfying a given property, without passing through an area of points satisfying another property.

2.1 About spatial logics

Spatial logics enjoy a long tradition in the field of Mathematical Logics [16], and have been proposed as descriptive languages in Artificial Intelligence. We focus on a particular kind of modal spatial logics, whose intended models are both continuous structures (topological spaces) and discrete (graphs, digital images). In this section we recall some basic concepts of spatial logics. It is important to recall that spatial logics can employ any logical syntax: their common characteristic is that the structure of interpretation feature geometrical entities and relations.

Thus, any spatial logic is characterized by the following parameters:

- a logical syntax;
- a signature of geometrical primitives;
- a class of interpretations.

A first example of spatial logic was provided by Tarski [18]. In this logic, the object of reasoning are the points of a topological space. Tarski proposed to interpret each formula of the modal logics variant called S4 as a set of points. Atomic propositions denote arbitrary sets; boolean operations are translated to set-theoretical operations; finally, the modal operator $\diamond$ is the topological closure.

However, one can employ more complex structures as a class of interpretation, such as graphs. A different take on spatial logics was given in the
seminal work of Egenhofer [20], where formulas are interpreted on the class REGC of regular closed algebras of topological spaces. Thus, formulas in this logics can be used to specify properties about any topological space.

Egenhofer’s language, known as RCC8, was later enriched with boolean operators [21].

While these logics adopt a FOL syntax, modal logics are also suitable to specify properties about space. Some work has been done in this direction, for instance in [22]. The logic considered here moves in this direction, combining CTL operators with spatial ones and providing a Surrounded operator from which we can derive reachability.

2.2 Closure Spaces

In order to provide an interpretation of modal logics which is akin to the work of Tarski, and deals with finite, discrete structures such as graphs, it is necessary to generalise topological spaces. A natural notion of closure of a set of nodes in a graph adds to such set all the nodes that can be reached in one step. Such modal operator is not idempotent, therefore it cannot be modelled using topological spaces.

In [19], Closure Spaces are proposed as a generalization of topological spaces in order to deal with discrete structures. We recall here some basic notions about topological spaces, before introducing closure spaces.

Definition 2.1. A topological space is a pair $(X, O)$, where $X$ is a set and $O \subseteq \mathcal{P}(X)$ is a collection of subsets of $X$ called open sets, closed under arbitrary unions and finite intersections and such that $\emptyset, X \in O$.

Definition 2.2. In a topological space $(X, O)$, $A \subseteq X$ is closed if its complement is open.

Definition 2.3. In a topological space, the closure of $A \in X$ is the least closed set containing $A$.

An alternative definition is the one given by Kuratowsky:

Definition 2.4. A topological space is a pair $(X, \mathcal{C})$ where $X$ is a set and the closure operator $\mathcal{C} : \mathcal{P}(X) \rightarrow \mathcal{P}(X)$ assigns to each subset in $X$ its closure, obeying to the following laws for all $A, B \in X$:

- $\mathcal{C}(\emptyset) = \emptyset$
- $A \subseteq \mathcal{C}(A)$
\[ C(A \cup B) = C(A) \cup C(B) \]
\[ C(C(A)) = C(A) \text{ (idempotency)} \]

In this case, the closure operator is assumed to be a primitive one. However, due to the interderivability of closure and open set, these definitions end up to be equivalent. Indeed, to obtain the Kuratowski definition from topological space, we can define the closure operator as the topological closure. The obtained closure operator respects the properties of Kuratowski definition. On the other hand, starting from a Kuratowski topological space, we can define the open sets as the sets \( A \) s.t. \( A = \overline{C(A)} \).

As said, closure spaces generalize topological ones. Their definition is built on the previous one, by removing the idempotency property.

**Definition 2.5.** A closure space is a pair \((X, C)\), where the closure operator \( C \) satisfies all the properties of 2.4, except that idempotency.

We also recall the definition of interior, boundary and neighbourhood, which will be useful in the definition of spatial logics operators.

**Definition 2.6.** Let \((X, C)\) be a closure space:

- the interior of \( A \in X \), \( I(A) \) is the set \( C(A) \)
- \( A \) is a neighbourhood of \( x \in X \) if \( x \in I(A) \)
- \( A \) is closed if \( A = C(A) \) open if \( A = I(A) \)

### 2.3 Quasi-discrete closure spaces

We can derive closure spaces starting from a binary relation: in this case we talk about quasi-discrete closure spaces. These are of primary interest for our purposes, as every graph induce a quasi-discrete closure space, hence including graph underlying an image. We briefly introduce here this spaces and their properties.

**Definition 2.7.** Consider a set \( X \) and a relation \( R \subseteq X \times X \). A closure operator is obtained from \( R \) as \( C_R = A \cup \{ x \in X \mid \exists a \in A. (a, x) \in R \} \).

**Lemma 2.1.** The pair \((X, C_R)\) is a closure space.

This gives us the possibility of defining interior and boundary operators in terms of this relation:
Proposition 2.1. Given $R \subseteq X \times X$, in the space $(X, C_R)$, we have:

- $\mathcal{I}(A) = \{x \in A \mid \exists a \in \overline{A}. (a, x) \in R\}$
- $\mathcal{B}^{-}(A) = \{x \in A \mid \exists a \in \overline{A}. (a, x) \in R\}$
- $\mathcal{B}^{+}(A) = \{x \in A \mid \forall a \in A. (a, x) \in R\}$

2.4 Syntax and Semantics of SLCS

The logic SLCS defined in [17, 6] extends the classical (Tarski-style) topological semantics of S4 in two directions: first, its models are Closure Spaces, and second, it features a “surrounded” operator, which is based on reachability (and reachability can be derived from it), and a “propagation” operator, which essentially is backwards reachability (useful when the models are based on directed graphs). Here, formulas may predicate the fact that we can reach or be reached from points satisfying a given property by following a path in the space. Three spatial operators are provided:

- a one step operator, denoted by $N$;
- a propagation operator, denoted by $P$;
- a surrounded operator, denoted by $S$;

The syntax of SLCS is given by the following grammar:

$$
\langle \Phi \rangle ::= a \quad [\text{Atomic proposition}]
| \top \quad [\text{True}]
| \neg \Phi \quad [\text{Negation}]
| \Phi \land \Phi \quad [\text{And}]
| N\Phi \quad [\text{Near}]
| \Phi S\Phi \quad [\text{Surrounded}]
| \Phi P\Phi \quad [\text{Propagation}]
$$

Before providing the SLCS semantics, we need to introduce the definition of a closure model in which we can define satisfaction of a formula.

Definition 2.8. A closure model is a pair $\mathcal{M} = ((X, C), \mathcal{V})$ consisting of a closure space $(X, C)$ and a valuation $\mathcal{V} : X \rightarrow 2^{AP}$, assigning each point to the set of atomic proposition it satisfies.
Digital images as closure models  It is thus easy to see that digital images are a suitable model for SLCS: indeed, as the underlying structure is a planar graph, hence a closure space. The valuation function $V$ is given by a pixel’s color properties, as e.g. its intensity or components.

We can now give the semantics of the logic in form of satisfaction of formulas:

**Definition 2.9.** Satisfaction $\mathcal{M}, x \models \phi$ of a formula $\phi \in \Phi$ at point $x \in X$ in model $\mathcal{M} = ((X, C), \mathcal{V})$ is defined by induction on the structure of terms by the following equations, where $p : x \rightsquigarrow \infty$ is a path from $x$ (i.e. $p(0) = x$) and $p : x \rightsquigarrow^i_x \infty$ is a path from $x$ with $p(i) = y$:

$$\begin{align*}
\mathcal{M}, x \models a \in AP & \iff x \in \mathcal{V}(a) \\
\mathcal{M}, x \models \top & \iff \text{true} \\
\mathcal{M}, x \models \neg \phi & \iff \mathcal{M}, x \not\models \phi \\
\mathcal{M}, x \models \phi_1 \land \phi_2 & \iff \mathcal{M}, x \models \phi_1 \land \mathcal{M}, x \models \phi_2 \\
\mathcal{M}, x \models N \phi & \iff x \in C(\{y \in X \mid \mathcal{M}, y \models \phi\}) \\
\mathcal{M}, x \models \phi_1 S \phi_2 & \iff \mathcal{M}, x \models \phi_1 \land \forall p : x \rightsquigarrow \infty. \forall l. \mathcal{M}, p(l) \models \phi_1 \\
& \quad \implies \exists k. 0 < k \leq l. \mathcal{M}, p(k) \models \phi_2 \\
\mathcal{M}, x \models \phi_1 P \phi_2 & \iff \mathcal{M}, x \models \phi_2 \land \exists y, p : y \rightsquigarrow^l_x \infty. \mathcal{M}, y \models \phi_1 \land \\
& \quad \forall i. 0 < i < l \implies \mathcal{M}, p(i) \models \phi_2
\end{align*}$$

While boolean and $N$ operators behave as expected, other spatial operators need some further explanations.

Intuitively, $S$ has the meaning of being in a point which satisfies property $\phi_1$, there is no way to escape from property $\phi_1$ unless one pass from at least a point which satisfies $\phi_2$. It becomes clear that this operator can be used to derive reachability, as we are going to see.

As for $P$, the intuitive meaning is that, given a point that satisfies $\phi_2$, such a point can be reached from another one that satisfies $\phi_1$ by only traversing points which also satisfy $\phi_2$.

We now introduce some derived operators, which are all available as Voxel-LogicA’s primitives.
\[ \bot \equiv \neg T \quad \phi_1 \lor \phi_2 \equiv \neg (\neg \phi_1 \land \neg \phi_2) \]
\[ (I \phi) \equiv \neg (N \neg \phi) \quad \delta (\phi) \equiv (N \phi) \land \neg (I \phi) \]
\[ \delta^- (\phi) \equiv \phi \land \neg (I \phi) \quad \delta^+ (\phi) \equiv (N \phi) \land \neg \phi \]
\[ \phi_1 R \phi_2 \equiv \neg ((\neg \phi_2) S (\neg \phi_1)) \quad E \phi \equiv \phi S \bot \]
\[ F \phi \equiv \neg E \neg \phi \quad \phi_1 A \phi_2 \equiv \neg (\phi_1 P (\neg \phi_2)) \]

**Proposition 2.2.** Satisfaction of the derived operators is given by the following equations:

\[ \mathcal{M}, x \models \phi_1 R \phi_2 \iff \exists p : x \rightsquigarrow \infty, k. \mathcal{M}, p(k) \models \phi_2 \]
\[ \land \forall j. 0 < j \leq k \implies \mathcal{M}, p(j) \models \phi_1 \]
\[ \mathcal{M}, x \models \phi_1 A \phi_2 \iff \mathcal{M}, x \models \phi_2 \lor \forall y. \mathcal{M}, y \models \phi_1, \forall p : y \rightsquigarrow \infty \]
\[ \exists i. 0 < i < l \land \mathcal{M}, p(i) \models \phi_2 \]
\[ \mathcal{M}, x \models E \phi \iff \forall p : x \rightsquigarrow \infty \land i \in \mathbb{N}. \mathcal{M}, p(i) \models \phi \]
\[ \mathcal{M}, x \models F \phi \iff \exists p : x \rightsquigarrow \infty \land i \in \mathbb{N}. \mathcal{M}, p(i) \models \phi \]

Let us focus on the reachability operator \( R \). A point \( x \) satisfies \( \phi_1 R \phi_2 \) if and only if \( \phi_2 \) holds in \( x \) or there is a path whose points all satisfy \( \phi_1 \) and lead to a point satisfying both \( \phi_1 \) and \( \phi_2 \) (except for \( x \) itself, which may satisfy \( \phi_1 \) or not).
3 VoxLogicA

VoxLogicA is a model checker for an extended superset of the SLCS primitives, which operates only on 2D and 3D digital images. Formulas are interpreted on points, called pixels in 2D images and voxels in 3D images. VoxLogicA is written in F#, and its primitives are executed in-CPU, mostly using the state-of-the-art imaging library SimpleITK ¹.

Each pixel/voxel is treated like a node in a graph, thus allowing to use a model checking approach based on SLCS. Arcs of the graph are based on the proximity between nodes, namely two voxels are connected if they share a vertex (therefore, in a 2D image, a non-borderpoint has 8 neighbours, whereas in a 3D image, a non-border point has 26 neighbours).

In order to specify properties on images, VoxLogicA is provided with an interpreter for a declarative language, called Image Query Language (ImgQL in short), which includes all the boolean and modal SLCS operators plus operators for manipulating images, such as thresholding or RGBA components selection.

The embedded model checker uses an explicit state approach, meaning that the state descriptor for a system is maintained explicitly, as well as all state transitions. Of course, this can lead to an exponential growth of the search space. However, as VoxLogicA considers only pre-built models, this growth is limited and the checker can deal with pretty big systems. Furthermore, the inherent redundancy of this kind of approach is actually mitigated by the fact that the considered images must be fully explored (colored).

3.1 Project structure

VoxLogicA is a research project written in F# ² using the dotnet core run time ³. As a functional language, F# is apt to write the interpreter and model checking core of VoxLogicA. Dotnet language interoperability allows for using any library which is available in the dotnet ecosystem, without having to create new bindings.

In Section 3.2 we will discuss the input language of VoxLogicA, derived from the logic SLCS. Here we analyse the structure of the project in order to better understand which of its parts have been changed in the GPU based version.

¹See https://simpleitk.org
²See https://fsharp.org/.
³See https://dotnet.microsoft.com/.
Operators  Operators are defined in a set of abstract interfaces dividing them into subsets that are meaningful fragments of the language (e.g. spatial operators, imaging operators, etc.). A specific implementation of VoxLogicA (such as the one that operates in-CPU) needs first of all to introduce concrete types, implementing the abstract interfaces, and then to provide algorithms to implement the operators. Interfaces and operators signatures are summarized in fig. 1.

These operators require different parameters type. Types in VoxLogicA are the following ones:

- model
- number
- bool
- string
- valuation

The model type serves as the input type: each input image is indeed a model. The valuation type is a “wrapper” type, which encapsulates number or bool and represents the result of boolean or numerical operations on a model.

Explicit state model checking via dynamic programming  The model checker uses a dynamic programming approach which exploits memoization, i.e. the storage of partial results whose purpose is to avoid multiple evaluations of the same expression. In this kind of approach, we usually build a vector whose indexes are subformulas: each element of the vector contains the result of evaluating a subformula. This vector acts as a cache for avoiding to recompute the same formula several times.

In VoxLogicA, things are quite different. Memoization is “precomputed” during the parsing phase performing hashing on the syntax tree: base formulas are associated with a value (e.g. the filename for a load operation or an integer for numeric constants), while internal nodes are associated with a tuple $(function\_name, hash(parameter_1),...,hash(parameter_n))$. Each time we find a new formula, we assign a new (zero-based) numeric id, which will be used as the subformula’s index in the vector.
<table>
<thead>
<tr>
<th>ILogicModel</th>
<th>IImageModel</th>
<th>IStatisticalModel</th>
<th>IDistanceModel</th>
<th>ISpatialModel</th>
</tr>
</thead>
<tbody>
<tr>
<td>Intensity</td>
<td>CrossCorrelation</td>
<td>DT</td>
<td>Near</td>
<td>Through</td>
</tr>
<tr>
<td>Red</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Green</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Blue</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Alpha</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>RGB</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>RGBA</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Volume</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>MaxVol</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Percentiles</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>LCC</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>IQuantitativeModel</td>
<td>IQuantitativeModel</td>
<td>IBoundedModel</td>
<td>IBooleanModel</td>
<td></td>
</tr>
<tr>
<td>Const</td>
<td>Border</td>
<td>BConst</td>
<td>TT</td>
<td></td>
</tr>
<tr>
<td>(Eq-Leq-Geq)SV</td>
<td></td>
<td></td>
<td>FF</td>
<td></td>
</tr>
<tr>
<td>Between</td>
<td></td>
<td></td>
<td>Not</td>
<td></td>
</tr>
<tr>
<td>Abs</td>
<td></td>
<td></td>
<td>And</td>
<td></td>
</tr>
<tr>
<td>Max-Min</td>
<td></td>
<td></td>
<td>Or</td>
<td></td>
</tr>
<tr>
<td>SubtractVV</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>AddVV</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>MultiplyVV</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Mask</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Avg</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>(Add-Mul)VS</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>(Sub-Div)VS</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>(Sub-Div)SV</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 1: Overview of the ILogicModel interface
Thus, once the checker starts, each subformula gets an id. Each evaluation is executed as a separate job, namely a concurrent task. Indeed, each operation in the ILogicModel interface returns a job containing a handle to the partial result of the evaluation. The overall truth value is built bottom up from the results of the evaluation of the subformulas.

The highly modular structure of VoxLogicA allows for providing different concrete implementations of the ILogicModel interface by replacing two files, SITKUtil and SITKModel, which provide the standard CPU implementation based on the SimpleITK library\(^4\).

All the interfaces are parameterised over a type named 'Value, and the return type of operations is job('Value), permitting implementations to define their own type for evaluation. In the CPU implementation of VoxLogicA, 'Value is instantiated to a simple data type denoting numbers, Boolean values and images.

The GPU implementation, as we shall see, is more complex, and 'Value is not just a simple data type, but rather a handle of a computation which is run in GPU, so that multiple computations can be composed from the main program running in CPU, without ever exchanging data between the CPU and the GPU (this would be inefficient).

The SITKUtil module, has several methods representing the concrete implementation of the ILogicalModel operators that can be called from SITK-Model to be encapsulated in a job. Thus, to change our implementation we need a new “handle type” and a different implementation for each module. In Section 4 we are going to see how these two modules have been replaced.

### 3.2 Image Query Language

Image Query Language (ImgQL, in short) is a simple declarative language allowing for writing formulas in a simple and concise way. Operators provided by the language can be summarized as follows:

- utility operators (load/save/print);
- images manipulation operators (intensity, thresholding, etc.);
- boolean operators;

\(^4\)Indeed, one could as well dynamically load the implementation from a pre-compiled assembly; this is entirely possible within the dotnet core architecture, even though it was not yet needed in VoxLogicA.
• modal operators.

These are defined in the stdlib.imgql file, while other operators can be defined by the user. In Section 5 we will see an example of an ImgQL program.
4 GPU based implementation

4.1 GPGPU programming

GPUs have been initially developed as dedicated computation units, only devoted to graphic purposes. Mid and high end computers usually come equipped with a graphic card, which in turn is equipped with a GPU. During the years graphical improvements in software (especially in games and 3D modelling and animation software) caused a quick improvement in GPUs performances. While early GPUs were basically graphic pipelines, they quickly evolved into more complex architectures.

In the early 2000’s, parallel computing has grown in importance, due to the physical limitation in increasing the computational power of a single core. While CPUs and GPUs switched from a single core to a multi core architecture, it became clear that one can have two main forms of parallelism:

- **stream parallelism**, as we have for instance in pipelines: in this case, different functions are applied over a data flow, adequately partitioned;

- **data parallelism**, as we have for instance in a parallel for: in this case, the same operation is performed over small pieces of data.

GPUs fit well the second case [23]. In the last fifteen years, then, some programming languages to exploit these devices for general purpose computing have been developed [24]. The most popular ones are NVidia’s CUDA C++ (2007), a proprietary language based on C++ included in the CUDA ecosystem\(^5\), and OpenCL\(^6\) (2008), a Khronos Group open standard project based on the C language which can be used on any GPU (differently from CUDA, which can be used only on NVidia devices). Both languages provide constructs for transferring data from the host to the device memory and viceversa, along with constructs to launch functions to be executed on GPU (called kernels) and various synchronization mechanisms, both on the host and on the device side.

4.2 GPU architecture

We recap here some basic notions about GPUs, in particular about their memory and execution model, which are of primary importance to understand how to manage operations on these devices. We use here the OpenCL

---

\(^5\)https://developer.nvidia.com/about-cuda
\(^6\)https://www.khronos.org/opencl/
As said, a GPU has hundreds of cores. Cores are grouped on separated dice, each one having its own local (shared) memory. Local memory is quite fast (even if it is slower of a core's registers) and can be used to improve performances. On the programmer side, each core is a work-item (thread) executing the kernel code. Threads are grouped into workgroups and can synchronize over the shared memory.

The memory hierarchy also include a constant memory, a read only memory containing constants, and a global memory, which is accessible by the host and it is then used to pass code and data to the device and viceversa. Global memory is the slowest one in the hierarchy, thus caching data in the local memory can lead to better performances (cached memory access). Furthermore, due to this particular architecture, accesses to global memory must be coalesced, i.e. threads in a workgroup should access consecutive locations in memory and the first location to be accessed should be a multiple of the workgroup size.

Another peculiarity is the fact that different execution paths are serialized on GPUs. Consider, for instance, an if-then-else statement: in this case, threads evaluating to true will execute the then branch, while threads evaluating to false are disabled. Once the former computation finishes, threads evaluating to false will run the else branch. This phenomenon, called thread divergence, obviously reduces the achieved parallelism and can quickly degrade the overall performances.

Finally, it is important to remark how a GPU program must be set up and
executed. Compilation and building of the program are executed at run time by the calling program. Kernels and a command queue are then created. Kernels are pushed into the queue in the right order, unless one specify an out-of-order execution: in this case it is necessary to use special constructs, called events, to enforce the correct execution order. Programmers can also push barriers and reads on the queue, which are used to reach a certain point of the computation ensuring that all the previous kernels have been executed and to get back the result of a computation from the GPU.

4.3 About the comparison between GPUs and CPUs

There is a lot of discussion about the possibility of comparing performances of GPUs and CPUs programs, due to the extremely different architectures and purposes of these devices [25, 26]. When a CPU and a GPU are actually comparable? Is it fair? Does it make any sense?

From this thesis perspective, the comparisons we will make between the performance of GPU-based and CPU-based implementation should only be considered as indications that this kind of task is indeed suitable for GPUs and that the chosen architecture effectively exploits a GPU’s capabilities. Of course, the decision to switch to a GPU based implementation, especially for an already optimized, well performing CPU project, should be supported by some basic considerations about the main characteristics of the program. In particular, moving to GPUs should be taken into account when, as said, the main tasks are highly data parallel and the input granularity is “big enough” to exploit the huge degree of parallelism without suffering the extra-work needed in order to set up the computation, as stated by the Gustafson’s Law [28]. Indeed, as we compute some tasks, we can divide the workload $W$ as follows:

$$\alpha W + (1 - \alpha)W$$

where $0 \leq \alpha \leq 1$, $\alpha W$ is the serial fraction of the code and $(1 - \alpha)W$ is the non-serial fraction. As we scale the workload on $n$ cores, we get:

$$W(n) = \alpha W + (1 - \alpha)nW$$

which in turn gives us the following speedup in latency:

$$S_n = \alpha + (1 - \alpha)n$$

meaning that only the non-serial part will enjoy a real speedup as computational resources increase. Thus we get that parallelism is actually exploited as we process large problems.
Furthermore, one should not consider the two kinds of approach as mutually exclusive. In a typical setting, CPU and GPU can cooperate, moving the most computationally intensive tasks to the GPU and leaving the rest to the CPU, thus considering the so called *heterogeneous computing* in order to exploit each device’s characteristics in a convenient way [27].

### 4.4 VoxLogicA GPU based implementation

We discuss here the implementation details and choices for moving VoxLogicA to GPUs. In particular, most of the work has been done in order to set up properly the computation, still maintaining the original VoxLogicA structure. To this end, we replaced the files *SITKModel* and *SITKUtil* with two new modules, named *GPUModel* and *GPUUtil*, implementing the same interface and providing the GPU version of all the VoxLogicA operations. The GPU code is included in the *kernel.cl* file. In order to manage GPUs operations, we use the CLoo library, a C# library providing dotnet bindings for OpenCL.

**GPUImage** The GPUImage type is a simple utility type that acts like a wrapper for the OpenCL image buffers, carrying along with it some additional information. More precisely:

- the *BaseImg* property is the image buffer;
- the *BaseComps* property contains the number of components of the image;
- the *BaseType* property contains the pixel value type.

For the time being, VoxLogicA supports R (monochromatic) and RGBA (red, green, blue, alpha) images, whose pixel type is `uint8`, `vector-uint8`, `uint16`, `uint32`, `float32`.

```csharp
1    interface IBooleanModel<GPUImage> with
2        member __.TT = job {
3            return handler.Const (getBaseBuffer(), events, queue,
4                (List.filter (fun (x : ComputeKernel) -> x.
5                  FunctionName = "trueImg") kernels))
6        member __.FF = job {
```

---

7 All the source code for this thesis is public, and available in the VoxLogicA repository [https://github.com/vincenzoml/VoxLogicA](https://github.com/vincenzoml/VoxLogicA).

8 See [https://github.com/clSharp/CLoo](https://github.com/clSharp/CLoo).
return handler.Const (getBaseBuffer(), events, queue, (List.filter (fun (x : ComputeKernel) -> x. FunctionName = "falseImg") kernels)) }

member __.BConst value = job {
if value then
  return handler.Const (getBaseBuffer(), events, queue, (List.filter (fun (x : ComputeKernel) -> x. FunctionName = "trueImg") kernels))
else
  return handler.Const (getBaseBuffer(), events, queue, (List.filter (fun (x : ComputeKernel) -> x. FunctionName = "falseImg") kernels))
}

---

**Listing 1: GPU buffers handling in GPUModel**

**GPUModel** The GPUModel module is responsible for setting up the GPU computation. Compilation and building of the OpenCL program, as well as the creation of the Events list and of the Command Queue are executed in the GPUModel constructor, as it is necessary to provide them to the GPUUtil module, which actually invokes the GPU code.

Mirroring the standard VoxLogica implementation, each operation is launched as a job (i.e. a separate process), except for the `Load`, `Save` (which is partially demanded to the GPUUtil module) and `CanSave` operations. In particular, during the Load operation, we check if the input image type is supported and we create a GPUImage containing the image buffer and the information about the image type and components. In this way, processing an image whose type is not supported will immediately cause the process to return.

In order to keep the pipeline full during computation, all the kernels are created within the GPUModel constructor. In order to retrieve the correct one, we filter the kernel list providing the function name.

It is worth noting that we cannot make any assumption about the input formulas, the image and the correct execution order, thus the operations must be fully composable. This is coherent with the algebraic perspective, even if it complicates memory management. Indeed, each time an operation is invoked, we get back from the GPU a handle pointing to the buffer containing the resulting image, in such a way that this can be provided as an input to the subsequent operations. It is worth noting that images are never copied from the GPU to the CPU, except that in the Save operations. In particular, intermediate results, that are not explicitly saved, are never copied back.
member this.UOp (img : GPUImage,
    events : List<ComputeEventBase>,
    queue : ComputeCommandQueue,
    kernel : list<ComputeKernel>) =

    let outformat = ComputeImageFormat(img.BaseComps, img.
        BaseType)

    let obuf = new ComputeImage2D(context,
        ComputeMemoryFlags.ReadWrite ||
        ComputeMemoryFlags.AllocateHostPointer,
        outformat,
        img.BaseImg.Width,
        img.BaseImg.Height,
        0L,
        IntPtr.Zero)

    kernel.[0].SetMemoryArgument(0, img.BaseImg)
    kernel.[0].SetMemoryArgument(1, obuf)
    queue.Execute(kernel.[0],
        null,
        [|int64 img.BaseImg.Width;
        int64 img.BaseImg.Height|],
        null,
        events)

    GPUImage(obuf, img.BaseComps, img.BaseType)

Listing 2: Managing kernels in GPUUtil

GPUUtil  The GPUUtil module contains the actual definitions of the Vox-
LogicA operations. As many kernels have similar parameters and can be
discriminated from the GUPModel module, code has been grouped in order
avoid redundancy. Thus we have basically three kind of operations:

- Unary operations;
- Image-image binary operations;
- Image-scalar binary operations.

We then have other operations for extracting components, creating constant
images (both numeric and boolean) and for computing reachability (Through
operation).

The GPUUtil module is the one interacting with the GPU during the com-
putation. As previously said, operations have to be fully composable: it is
thus necessary to check the type of the input image and to provide a coher-
ent output: in order to do that, we take GPUImage objects as inputs and
provide a new one as an output, setting adequately the objects properties.
A fundamental difference between F# and OpenCL is the type system. Recall that OpenCL is based on the C language, thus being weakly statically typed. Furthermore, GPUs do not fully support “classical” types, (for instance, many of them do not fully support doubles), and provide vector types, which are basically tuples and can be accessed in a similar way.

On the other hand, F# uses type inference (even if one can explicitly provide a name’s type) and provides a different type hierarchy. Then it has been necessary to map the F# types into the OpenCL types, ensuring no information loss: in particular, on the OpenCL side we mainly use unsigned int and float, which have both 32-bit precision. As supported pixel types are at most 32-bit unsigned integer or float, this prevents any information loss, while moving to 64-bit precision would dramatically affect performances.

```c
__kernel void termination(__read_only image2d_t inputImage1,
                        __read_only image2d_t inputImage2,
                        __global unsigned int* results,
                        __local unsigned int* tile) {
int2 gid = (int2)(get_global_id(0), get_global_id(1));
int2 lid = (int2)(get_local_id(0), get_local_id(1));
int2 gr_size = (int2)(get_local_size(0), get_local_size(1));
int x = gid.x;
int y = gid.y;
int2 size = get_image_dim(inputImage1);
const sampler_t sampler =
    CLK_NORMALIZED_COORDS_FALSE | CLK_ADDRESS_CLAMP |
    CLK_FILTER_NEAREST;
uint4 base = read_imageui(inputImage1, sampler, gid);
uint4 comp = read_imageui(inputImage2, sampler, gid);
tile[lid.x + lid.y*gr_size.x] = (base.x == comp.x);
results[get_group_id(0) + get_group_id(1)*get_num_groups(0)] = 0;

//Reduce rows
for(uint stride = gr_size.x/2 - 1; stride > 0; stride /= 2) {
    barrier(CLK_LOCAL_MEM_FENCE);
    if(lid.y < stride) {
        tile[lid.x + lid.y*gr_size.x] *= tile[(lid.x) + (lid.y + stride)*gr_size.x];
    }
}

//Reduce first column
```
if(lid.x == 0 && lid.y == 0) {
    for(uint i = 0; i < gr_size.y; i++) {
        tile[lid.x + lid.y*gr_size.x] *= tile[i];
    }
    results[get_group_id(0) + get_group_id(1)*
            get_num_groups(0)] = tile[0];
}  

Listing 3: GPU reduce

Kernels  Kernels definition obviously mirrors the operations of VoxLogicA.
OpenCL provides support for image types, thus inputs have read only image
type and outputs have write only image type. It is possible to read a pixel
using the read_imageui or read_imageuf functions: as the return types of
these are vector types (uint4 and float4), we can store images having up to
4 components. Writing images can be done using the write_imageui and
write_imageuf functions.
Depending on the declaration on the host side, OpenCL pads memory in
order to optimize access: should there be unused components, the read and
write function will automatically set them to 0.
In order to maintain compositionality, operations are quite fine grained: we
thus have several kernels performing simple operations such as logical oper-
ations or arithmetic ones. Code has been written in such a way that we avoid
branches as much as possible, exploiting arithmetic operations and internal
representation of boolean (which are indeed integers) to replace conditional
statements.
Hence, the majority of the kernels do not need to be optimized using lo-
cal memory or synchronization, but only to exploit constant memory when
dealing with read only scalar values. It is the case, for instance, of the
thresholding kernels, where we transform an image from grayscale to b/w.
The threshold value is here stored in the constant memory, from where it is
read and stored into a register in order to allow faster access.
From an optimization’s perspective, the most interesting code is the one
written for computing volume, min/max and connected components la-
belling termination. In this case, we basically need to perform a reduce,
an operation which uses an associative function in order to combine ele-
ments in a matrix and get a single value as an output. We can thus exploit
workgroups and local memory to improve performances, instead of accumu-
lating the result directly into the global memory.
In order to do that, we need to declare a workgroup size (recall that this must be a divisor of the problem size) and the number of workgroups. On the “host” (that is, the CPU) side, we can store an array whose size is the number of workgroups: this will store the results of the local computations. Kernels are provided with a local argument, basically claiming to keep a portion of local memory available to store the partial results.
In the kernel code, we need to retrieve a thread global id and local id, which identifies it w.r.t. the overall number of threads and threads in the same workgroup. We also get the local size, which must be used to correctly access local memory. Each thread performs a loop: iteratively it “adds” an element inside a certain radius in the same row to its partial result. The radius is halved at each iteration. Threads synchronize at the beginning of each iteration to avoid race conditions.
Finally, the first thread of each workgroup reduces the first column of the tile matrix and stores the local result into an array in the global memory. Once we read the array on the host side, we can finalize the reduction on CPU simply appending (either summing or multiplying) the partial results. Thus, given an $n \times m$ matrix, the overall complexity is $\theta(n\log m)$.

**Connected Components Labelling**  As seen in the previous section, CC labelling is a core operation in VoxLogicA, as it can be exploited for computing reachability. It is thus important to have an efficient implementation for it, as due to its complexity, it can easily become a bottleneck.
On GPUs, algorithms for CC labelling can be classified as follows:

- *iterative algorithms*. In this case, a kernel “colors” a pixel according to its closest neighbourhood (the 8 direct neighbours in our case, as we employ 8-connectivity) and it is iteratively called by the host. In this way, colors are propagated globally until the input and the output images coincide;

- *direct algorithms*. Here a kernel computes labels locally. Its output is a map of local labels. From this map, we can compute a set of equivalences between labels: basically, as long as two adjacent pixels have different, non zero labels, then their labels are equivalent. Finally, a kernel colors the pixels according to the equivalence classes.

It is easy to see that the first approach can easily become a bottleneck, due to the iterative calls from the host. Furthermore, checking termination is
expensive, even using reduce, and the number of iterations required in order to fully color an image can significantly vary between different images. However, as a first attempt, we chose to use an iterative approach. For simplicity, we have used the algorithm presented in [29]. It is based on cellular automata and can be summarized as follows:

- as a first step, we mark white pixels having their left, top-left and top neighbours black as left corners. The label is computed as follows:

\[ \sum_{i=0}^{d} c[i] \cdot s[i]^i \]

where \( d \) is the number of dimensions, \( c \) is the vector of coordinates and \( s \) is the vector of dimensions. This guarantees uniqueness of labels.

- the iterative step checks a pixel neighbourood: every time a neighbour’s label is greater than the pixel’s one, then the pixel’s label is replaced with the former.

- After some iterations, we check if the input and output images of the iterative step coincide. The algorithm stops when this check evaluates to true.

Preliminary tests have been performed on an NVidia GTX1050, a pretty outdated GPU, and on a 8 core AMD Ryzen for the CPU version. Here the global algorithm shows its limitations, running in circa 600ms, while the CPU version runs in circa 100ms. Thus we try to optimize the algorithm exploiting local memory, performing some iterations inside a kernel and avoiding some of the global calls. Kernels are shown in listing 4.4. This variation of the algorithm significantly improved performances on our system, reducing the gap with the CPU version. In the next section we are going to see more precisely our tests results. The local algorithm performs then as follows:

- Our first step, find corner stays untouched, as well as the global propagation step.

- After one global iteration, we propagate colors (labels) locally in a tile. As threads only synchronize locally, we cannot exploit local memory to propagate over a tile’s boundaries. Then we call again global propagation to communicate colors over tiles.
• Termination stays as in the global algorithm.

```c
__kernel void color_components(__read_only image2d_t image,
    __read_only image2d_t inputImage1, __write_only image2d_t outImage1) {
    int2 gid = (int2)(get_global_id(0), get_global_id(1));
    int x = gid.x;
    int y = gid.y;

    int2 size = get_image_dim(inputImage1);
    const sampler_t sampler = CLK_NORMALIZED_COORDS_FALSE |
        CLK_ADDRESS_CLAMP | CLK_FILTER_NEAREST;

    uint4 base = read_imageui(image, sampler, gid);
    int condition = base.x > 0;

    uint4 ui4a = read_imageui(inputImage1, sampler, gid);

    //propagate colors over tiles
    for(int a=-1; a<=1; a++)
        for(int b=-1; b<=1; b++) {
            uint4 tmpa = read_imageui(inputImage1, sampler, (int2)(x+a, y+b));
            if(tmpa.x > ui4a.x) {
                ui4a = tmpa;
            }
        }
    write_imageui(outImage1, gid, ui4a.x*condition);
}
```

```c
__kernel void propagate(__read_only image2d_t image,
    __read_only image2d_t inputImage1, __write_only image2d_t outImage1, __local unsigned int* tile) {
    int2 gid = (int2)(get_global_id(0), get_global_id(1));
    int2 lid = (int2)(get_local_id(0), get_local_id(1));
    int2 gr_size = (int2)(get_local_size(0), get_local_size(1));

    uint4 base = read_imageui(image, sampler, gid);
    int condition = base.x > 0;

    tile[lid.x + lid.y*gr_size.x] = read_imageui(inputImage1, sampler, gid).x;

    //propagate colors in a tile
```
for(int i = 0; i < 2; i++) {
    for(int a=-1; a<=1; a++) {
        for(int b=-1; b<=1; b++) {
            barrier(CLK_LOCAL_MEM_FENCE);
            if(lid.x + a >= 0 && lid.x + a < gr_size.x 
                && (lid.y + b) >= 0 && (lid.y + b) < 
                gr_size.y) {
                uint tmpa = tile[(lid.x + a) + (lid.y + 
                    b)*gr_size.x];
                if(tmpa > tile[(lid.x + lid.y*gr_size.x 
                    )]) {
                    tile[lid.x + lid.y*gr_size.x] = 
                        tmpa;
                }
            }
        }
    }
}
write_imageui(outImage1, gid, tile[(lid.x + lid.y*gr_size.x 
            )]*condition);

Listing 4: CC colors propagation
5 Test results

All tests run on a system equipped with an AMD Ryzen 7 CPU and an NVidia GTX1050 GPU. The AMD Ryzen (2018) processor is equipped with 8 cores with a base clock of 3.7 GHz, up to 4.3 GHz. The NVidia GTX 1050 GPU (2016) is based on the proprietary Pascal architecture. It is equipped with 640 CUDA cores, a 2GB GDDR5 frame buffer, with a memory speed of 7 Gbps. Boost clock is 1455 MHz.

We mainly performed two kind of tests. At first we compared the two implementations on large “sequential” formulas of the form \( f(g(h(...(x))) \), not including the Connected Components operator. In this setting, the GPU implementation exhibits a quite important speedup, which confirms our hypothesis that the “global” spatial model checking algorithm scales well when implemented in GPU.

Then, we used a more realistic formula, extrapolated from the case study in [8], which includes reachability, therefore labelling connected components. In this case, the performance of the connected components algorithm that we have implemented so far, possibly due to synchronous execution, is not in par with the CPU. This will require further investigation.

The images of test are grayscale images of size 1920x1920.

```plaintext
1 load imgPFLAIR = "./BRATS/Brats17_2013_17_1/
  Brats17_2013_17_1_pflair-uint16-alike_slice-max-GTV-
  uint16-scaled.png"
2 let f(x) = near(interior(x))
3 let g(x) = f(f(f(x))))
4 let h(x) = g(g(x)))
5 let i(x) = h(h(h(x)))
6 let k(x) = i(i(i(x)))
7 save "x.png" k(red(imgPFLAIR) >= . 62330)
```

Listing 5: Test code for large formulas

In order to test scalability of the GPU version, we first run a simple test were we nest formulas and we call the resulting one multiple times. An example formula is shown in listing 5. The aim of this test is mainly to show the effectiveness of memory management. Indeed, as we call different functions multiple times, undesired data movements between the host and the device would increase the total execution time.
This is not the case: as the size of the formula grows, the execution time grows linearly in both cases, but with a much smaller slope for the GPU version (fig. 2). The test has been repeated several times with different formulas, always obtaining similar results.

This confirms that the overhead of preparing GPU computation is well balanced by the memory management, which allows us to exploit the device’s parallelism and shows a good scalability of the GPU version. We can also observe that having small operations implemented as stand-alone kernels does not carry overheads, as long as the result of each operation is readily available to the following one.

Furthermore, we can expect even better results on 3D images, due to the 26-connectivity.

5.1 A case study: medical image analysis

Automated tools are of primary importance in medical image analysis. As this task can be very difficult to perform by hand, the rise of tools for image analysis has significantly eased it. Automated tools can be used to perform various tasks, as for instance computer-aided diagnosis, whose aim is to identify symptoms of a specific disease or image segmentation, whose
objective is to identify areas exhibiting certain features. Image segmentation is the task considered in this thesis case study for VoxLogicA, as it has already been a benchmark for the CPU version.

5.2 Image segmentation

Image segmentation is nowadays a pretty active research field. Both automatic and semi-automatic methods have been developed using, for instance, AI and optimization techniques as neural networks and swarm particle optimization. We focus here on brain tumor segmentation. In [8], a method for fully automated segmentation of brain tumors based on Spatial Logics, implemented via VoxLogicA, has been proposed. Accuracy of the results is in par with the state-of-the-art, and execution times are in the order of seconds. Other existing tools mostly rely on Neural Networks, however, one should not consider these two approaches as opposite, but rather complementary. Indeed, a neural network could be used to set up parameters in an analysis based on spatial logics. On the other hand, as medical image analysis require to follow certain protocols, one may think to use logics in order to check if Neural Network based tools are performing analysis respecting some specific constraints.

5.3 The BRATS benchmark

As a case study, we consider the BraTS Challenge (Brain Tumor Segmentation Challenge)\(^9\). The benchmark provide a set of tomographical brain images in which we want to locate a tumor. This kind of task is usually executed by hand, requiring time ranging from 30 minutes to some hours, since the expert needs to manually select voxels on each 2D "slice" of a

\(^9\)See http://braintumorsegmentation.org/
3D image, or in an automated way, using neural networks. In [8], a simple procedure has been proposed, consisting of:

- identifying two regions via thresholds, a "hyperintense" core which is known to be tumor, and a "very intense" area which may be tumor;

- use a reachability constraint, removing the very intense regions that do not touch a hyperintense region; these are considered noise;

- refining the result via texture similarity (using an algorithm that compares local histograms via cross-correlation).

```plaintext
import "stdlib.imgql"

// grow(a,b) is "a" union "those regions b that touch regions of a"
let grow(a,b) = (a|touch(b,a))

// Load data
load imgPFLAIR = "./BRATS/Brats17_2013_17_1/Brats17_2013_17_1_pflair-uint16-alike_slice-max-GTV-uint16-scaled.png"

// An image could have many attributes (e.g. RGB). We extract the intensity.
let pflair = intensity(imgPFLAIR)

// The actual analysis session starts here

// 1. Thresholding
let hI = pflair >. 62258 // (0.95 * 65535)
let vI = pflair >. 56360 // (0.86 * 65535)

// 2. Semantic noise removal via growing
let gtv = grow(hI,vI)

// Save the results
save "output/gtv.png" gtv
```

Listing 6: Brain tumor contouring

In this work, since we have not implemented local histogram analysis, we have extrapolated from the procedure in [8] the core method, including the aforementioned reachability constraint. Input to the model checker is shown in listing 5.3. Such procedure is used as a benchmark in the brain tumor
The CPU implementation of VoxLogicA has been tested on this benchmark with good results, which we want to compare to the ones obtained with its GPU based version.

In this case, the GPU labelling algorithm shows its limitations. In fig. 4 is shown the average execution time for the three implementations on the BRATS benchmark cases. Even if the GPU is slower in this particular case, it is worth noting that exploiting local memory carried a good improvement w.r.t. to the fully global version of the algorithm. This happens on our test system, where we can use at most a $6 \times 6$ workgroup (recall that the size of a workgroup must be computed as the largest integer that is smaller of the device max workgroup size and divides the problem size). Various experiments shown that the best we can do in this setting is to perform two local iterations before a global steps.

<table>
<thead>
<tr>
<th></th>
<th>CPU</th>
<th>GPU (global)</th>
<th>GPU (local)</th>
</tr>
</thead>
<tbody>
<tr>
<td>time</td>
<td>120-150</td>
<td>600-620</td>
<td>390-410</td>
</tr>
</tbody>
</table>

Figure 4: Connected components labelling execution time (milliseconds)

In order two understand how much the maximum workgroup size can affect the local connected components algorithm performances, we test it on an AMD Radeon M340. This is a notebook graphic card, providing 4 GB of DDR3 memory, with 384 shader cores and boost clock of 1021 MHz. Even if it cannot compete in performances with the GTX card, it allows for workgroups of size $12 \times 12$.

In this case, we can observe that the execution time of the algorithm significantly decreases: while the global algorithm runs in circa 2.8s, the local version runs in circa 600ms using the maximum available workgroup size and 4 local iterations (fig 5).
Figure 5: Local algorithm performances on growing workgroup size
6 Conclusions

The goal of the thesis has been substantially fulfilled, as we have demonstrated scalability and effectiveness of explicit-state spatial model checking on non-trivial images (size circa 4 megapixel), by obtaining a consistent speed-up on large formulas that do not use CC labelling. On CC labelling, so far, our best result has been that the the CPU is still 4 times faster than the GPU. The CC labelling algorithm in GPU remains a bottleneck in our current implementation. We note in passing that the implementation in SimpleITK is highly optimized, and it exploits run-length encoding to speed up CC labelling.

However, a number of factors make our numerical results more an indication for future work than an experimental validation. First and foremost, our tests have been conducted on an outdated, low end GPU. Our expectation is to have much better performances using a high end device, thus competing with the CPU even using this kind of algorithm.

Still, switching to a direct algorithm would significantly reduce the overhead of calling a kernel multiple times and testing for termination, and should be a priority as a future work.

Other refinements to the proposed infrastructure are left as future work: for instance, implementing other primitives (in particular local histogram processing) and supporting 3D images (which requires to develop dedicated kernels) possibly exploring a more hybrid solution, exploiting both CPU and GPU in a convenient way.

From a theoretical perspective, the main objective for the future is to develop an extension of SLCS to deal with dynamic graphs, possibly building on existing approaches, as the one proposed in [30]. This would allow to specify properties over graphs changing in time, thus covering a higher number of case studies.
References


