GRUEL: AN EL REASONER USING GENERAL PURPOSE COMPUTING ON A GRAPHICAL PROCESSING UNIT

MYRIAM KHARMA

A THESIS IN THE DEPARTMENT OF

COMPUTER AND SOFTWARE ENGINEERING

PRESENTED IN THE PARTIAL FULFILLMENT OF THE REQUIREMENTS

FOR THE DEGREE OF MASTER OF COMPUTER SCIENCE AT

CONCORDIA UNIVERSITY

Montreal, Quebec, Canada

April 2017

CONCORDIA UNIVERSITY
School of Graduate Studies

This is to certify that the thesis prepared

By: Myriam Kharma

Entitled: GRUEL: An EL Reasoner Using General Purpose Computing on a Graphical Processing Unit

and submitted in partial fulfillment of the requirements for the degree of

Master of Computer Science

complies with the regulations of the University and meets the accepted standards with respect to originality and quality.

Signed by the final examining committee:

_____ Chair
Dr. Adam Krzyzak

_____ Examiner
Dr. Dhrubajyoti Goswami

_____ Examiner
Dr. Sudhir Mudur

_____ Supervisor
Dr. Volker Haarslev

Approved by _____
Chair of Department or Graduate Program Director

_____
Dean of Faculty

Date _____

Abstract

GRUEL: AN EL REASONER USING GENERAL PURPOSE COMPUTING ON A GRAPHICAL PROCESSING UNIT

Myriam Kharma

The third installment of the worldwide web (Web 3.0) relies on an infrastructure of efficient reasoners, able to quickly infer new knowledge from existing data. We have created a reasoner using general purpose computing on a graphical processing unit (GPU) for the "Existential Language" description language, denoted with EL, that exploits the processing power of a GPU while attempting to solve a logic-based artificial intelligence problem.

To classify small ontologies, our system is nearly 70% faster than other reasoners we have compared it with. Currently far from perfect, GRUEL is planned to improve and expand, yet for the moment proves that better performance is achievable, and that general purpose computing on a GPU is a methodology worth exploring while developing the semantic web.

Acknowledgements

I had no idea how lucky I was in the Winter of 2015 when I shyly walked into Professor Volker Haarslev's office at Concordia University asking for his advice to enter the research program. Not only he accepted me into his team, but for the past two years, he also trusted me, advised me, pushed me to go further than my capabilities, and has been an academic role model ever since.

I thank, owe, and dedicate this thesis to the exceptional people I was honored to know and to learn from...

To Professor Volker Haarslev, who taught me everything I know in this field. With his priceless support, trust, encouragement, and endless knowledge, he made this work possible.

To Professors Dhrubajyoti Goswami, Leila Kosseim, Brigitte Jaumard, and Nematollaah Shiri, who each contributed indirectly to this thesis and taught me invaluable lessons. They each made my skillset grow in $O(e^t)$.

To Ms. Halina Monkiewitz and Professor Nancy Acemian, who taught me that stellar futures are made when we are dedicated to our job.

To my study mates and lab partners, Maryam, Nikoo, Zixi, Jelena, Razieh, Humaira, Ahmed, Ebtehaj, and Sanchit, who taught me that colleagues can sometimes be a complete support system.

To Ms. Bahareh Goodarzi, who taught me parallel design and behaved as a friend long before becoming one.

To the donors of Concordia University, namely those who donated for the bursary and awards I have been granted, and to every person who ever funded my education, who taught me that just like ambition, generosity should know no bounds.

To Professor Rached Zantout, who taught me to trust myself in the classroom.

To Mrs. Sonia Wehbe, who taught me to love the classroom.

And last but of course not least, to my mother, who taught me almost everything else.

'Logic takes care of itself;

All we have to do is to look and see how it does it.'

Ludwig Wittgenstein

Table Of Contents

List of Tables

List of Figures

1. Introduction

For the past few years, if not decades, two major trends have been becoming more popular in the domain of computer science. The first is artificial intelligence and the second is parallel and distributed programming. We present in this thesis a software architecture that we developed to solve a problem in artificial intelligence using parallel programming.

1.1 Scope and Objective of the Research

Our work is part of a "good old fashioned" approach to artificial intelligence that deals with knowledge. The term "good old fashioned artificial intelligence" is used in comparison to more recent methodologies in AI where statistical rules are employed. In the traditional procedure, a set of facts called a knowledge base is defined by humans. Once a knowledge base is developed, dedicated programs called reasoners apply preset rules to add new information in the form of axioms to the knowledge base. As knowledge bases grow, the need for faster and more efficient reasoners is growing too.

Our objective is to find ways to optimize reasoners and reduce their execution time. As with any recent performance optimization problem in programming, we can attempt to either parallelize the program, or implement more efficient algorithms. We have chosen to use parallel programming to implement a reasoner named "GRUEL" able to handle a limited vocabulary.

Particularly, we have chosen to exploit the hundreds of processors usually installed on a graphical processing unit in our parallel program. Doing so, we have utilized the processing power of a graphical processing unit to solve a general problem. This is called "general purpose computing on a graphical processing unit" and we now describe the particularity of our work.

1.2 Novelty of the Solution

To the best of our knowledge, at the time of writing this thesis, using general purpose computing on a graphical processing unit to implement a knowledge reasoner has never been attempted. Many reasoners exist that execute threads in parallel, but programming on a graphical processing unit is different from other forms of parallel computing due to the architecture of the device and the differences in its functionality and that of a central processing unit. That is why general purpose computing on a GPU is a discipline of its own, and it is for this reason that GRUEL is considered the first of its kind.

Achieving this goal was no easy task and we now describe the research procedure.

1.3 Research Procedure

We first started by reading literature about programming on a GPU. Having acquired the necessary knowledge and conducted a few practice experiments, we confirmed that our project is feasible, and we proceeded to the design phase, followed by the implementation. Often, we found gaps in our knowledge or errors in our assumptions and went back to the literature then repeated the design and amended the code. As we were building GRUEL from scratch and as we had an ambitious project that was not yet attempted, we had to do all the research and programming work ourselves. We could not rely on previously solved problems or on other people's experiences, but rather tried to find inspiration in solutions to problems that were slightly similar. This has deepened our knowledge of the discipline and enriched our experience. This document presents the summary of our work.

1.4 Outline

We start our discussion by exploring the frameworks behind building and reasoning on knowledge bases in Chapter 2. We then proceed by introducing graphical processing units and the particularities of general purpose computing using them. In Chapter 4, we describe GRUEL and the underlying data structures and algorithms. As with any software development project, we have

tested our system and compared its performance to other reasoners currently available. We discuss the results of these tests in Chapter 5. Finally, we describe the plan placed to improve this system in Chapter 6 and end the document with a conclusion.

2. Description Logics and the Semantic Web

2.1. Introduction

Artificial intelligence (AI) is divided into two major sub-parts, depending on the method the machine uses to acquire new knowledge. The first is called the Good Old Fashioned AI, where humans need to instruct the machines on how to come up with new information in the initial phase, and then the machine can deduce data on its own with an accuracy of 100%, provided the human input was correct. The second is commonly known as machine learning, and relies on probabilistic reasoning. In machine learning, the system learns on its own using practice data with confirmed results on one hand, and test data on the other. It calibrates itself and relies on statistical principles to predict the new knowledge based on historical findings initially in the practice data and in the system's logs. Its advantage is that human effort is quasi-absent beyond the programming of the system, yet its downside is that it may not always be accurate. Description logics fall into the Good Old Fashioned AI (GOFAI) category and have numerous applications. We discuss description logics in this chapter, namely some of their components and rules. We also describe one expressivity of description logics as well as different reasoning methods, and some applications of this work.

2.2 Description Logics

Good Old Fashioned AI is also known as logic-based AI. By logic, we mean predicate logic and it is used to model the world in what is known as *knowledge representation*. Computing systems can then use formal rules to reason on the knowledge at hand, and maybe come up with new information about the world represented. Description logics are a category of formal logic languages used in knowledge representation where unary and binary predicates are used to represent the domain [1]. Constructors and name instances are used and matched together to produce the model of the domain.

Inspired by philosophy, the term *ontology* in description logics designates a set of *axioms*, identifying the concepts in the domain and the relationships between individuals belonging to those concepts. In other words, an ontology maps a part of the real world using axioms, each of which being a definition of an item in the world. This definition may or may not include a relationship from an instance of an item, called individual, to another or to itself [12]. These items are the concepts in an ontology. For example, let us now model a small world composed of a nuclear family using plain English.

The Britannica Encyclopedia defines a "nuclear family" as a family made of two adults and at least one child socially recognized as theirs both [24]. We notice here that there are two conditions to make a group of people qualify as a nuclear family. First, the presence of exactly two adults and at least one child, and that the child must be the adopted or biological child of both parents. Adoption also can be a social norm and not necessarily legal. In addition, marriage is not restricted to individuals of different genders. Hence, in an ontology modeling human nuclear families, the below axioms hold and define possible relationships:

Listing 2.1: Axioms in the nuclear family ontology

1- A parent is an individual with at least one child
2- A parent can be either a father or a mother
3- A Father is a male person with at least one child
4- A mother is a female person with at least one child
5- Two persons might be married
6- A parent can either be a mother or a father
7- A daughter is a female child of an individual
8- A son is a male child of an individual
9- Two persons might be siblings
10- A sibling can be either a sister or a brother
11- A sister is a female sibling
12- A brother is a male sibling
13- A person who has a daughter is a person who has a child
14- A person who has a son is a person who has a child

In this example, we restrict genders to either male or female. To translate these axioms into description logics, we must first identify the concepts and the relationships. The relationships can also be called roles, or object properties. The following two tables list the concepts and the relationships found in this ontology.

Concept	**Definition**
Parent	Person with at least one child
Father	Male parent
Mother	Female parent
Daughter	Female child
Son	Male child
Sibling	Person with at least one sibling

Table 2.1 Concepts in the nuclear family ontology

Role	**Definition**
HasChild	Person has another person as a child
HasParent	Person has another person as a parent
HasSpouse	Person has a spouse, i.e. is in a marriage relationship with another person
HasSibling	Person has another person as a sibling.
HasDaughter	Has a female child
HasSon	Has a male child

Table 2.2 Roles in the nuclear family ontology

For conciseness, we do not include in this discussion the *has sister* and *has brother* roles, as they are both derivatives of *has sibling*. Similarly, we do not include *has wife*, *has husband, has mother, has father*. Working with these roles is identical to working with *has daughter* and *has son* from a description logics perspective.

To translate this ontology into a formal description logics language, we will use commonly known constructors introduced in [3]. We will first explore the constructors and the syntax, then write the description logics equivalents of the axioms in listing 2.1.

Constructor	**Definition**	**Example**	**Explanation**
≡	Equivalence	Person ≡ Child	A person is equivalent to a child
⊑	Subsumption	Father ⊑ Parent	A father is a parent, but a parent is not necessarily a father
¬	Negation	Male ≡ ¬ Female	Male and Female are mutually exclusive, i.e. disjoint.
⊓	Conjunction	Father ≡ Male ⊓ Parent	A father is a male AND a parent
⊔	Disjunction	Parent ≡ Mother ⊔ Father	A parent is either a mother or a father
⊤	Top concept	⊤ ⊑ Person	All concepts in the ontology are persons
∀	Universal restriction	ParentOfOnlyDaughters: ∀hasChild.Female	A *parent of only daughters* is an individual with all children female
∃	Existential restriction	Parent ≡ ∃hasChild.⊤	A parent is equivalent to an individual having at least one *hasChild* relationship with an individual of any other concept
⊥	Bottom concept	Leprechaun ⊑ ⊥	No leprechauns exist in the ontology
> ≥ < ≤	Number restrictions	Millipede ≡ Animal ⊓ ≥ 40 hasLegs	A millipede is an animal and has at least 40 legs

Table 2.3 Description logics constructors and definitions

Using the above constructors, we now translate the axioms in Listing 2.1 from natural English language to description logics.

English axiom	**DL equivalent**
A parent is an individual with at least one child	Parent ≡ ∃ hasChild.Child
A parent can be either a father or a mother	Parent ≡ Father ⊔ Mother
A Father is a male person with at least one child	Father ≡ Parent ⊓ Male
A mother is a female person with at least one child	Mother ≡ Parent ⊓ Female
Two persons can be married	Married ≡ Person ⊓ ∃ hasSpouse.Person
A parent can either be a mother or a father	Parent ≡ Mother ⊔ Father
A daughter is a female child	Daughter ≡ Child ⊓ Female
A son is a male child	Son ≡ Child ⊓ Male
Two persons can be siblings	Sibling ≡ Person ⊓ ∃ hasSibling.Person
A sibling can be either a sister or a brother	Sibling ≡ Sister ⊔ Brother
A sister is a female sibling	Sister ≡ Sibling ⊓ Female
A brother is a male sibling	Brother ≡ Sibling ⊓ Male
A person who has a daughter is a person who has a child	hasDaughter ⊑ hasChild
A person who has a son is a person who has a child	hasSon ⊑ hasChild

Table 2.4 DL equivalents of axioms in Listing 2.1

This list of axioms is not exhaustive. To properly design an ontology, we must consider that a system has no pre-acquired knowledge of the world. Thus, we must also include axioms that are

intuitive to us humans. For instance, we must define the Male and Female concepts, and we must make these two concepts disjoint using the negation constructor:

$$Male \equiv \neg Female$$

In addition, description logics contain role properties which we purposely omit for the moment as they are not yet relevant to our work. An example of an omitted role property is the relationship between *has child* and *has parent*. If a person is the child of another, then the latter is the parent of the former. Thus, *has child* and *has parent* are linked and the presence of one automatically causes the presence of the other. In this sense, this ontology is far from being faithful to the domain it models and does not contain all information held by a human regarding a nuclear family.

To facilitate working with ontologies, constructs are separated into sets of different description languages. We now discuss the one pertaining to our work.

2.3 The EL Expressivity

The EL description language is a limited vocabulary composed only of existential restriction, and conjunction. Below, we list the structure of the axioms that can be created in EL, using either concepts or roles. Items numbered 1 till 4 are presented in [6].

Listing 2.2: Axioms expressed with EL

1. $c1 \sqcap c2 \sqsubseteq c3$
2. $c1 \sqsubseteq c2$
3. $\exists r1.c1 \sqsubseteq c2$
4. $c1 \sqsubseteq \exists r1.c2$
5. $r1 \sqsubseteq r2$
6. $c1 \equiv c2$

In this listing, c1, c2, and c3 denote concepts, whereas r1 and r2 denote roles. Although the equivalence constructor is not among the constructors allowed in EL, but an equivalence can be expressed indirectly using two subsumption axioms. For instance, in the nuclear family ontology

that we have defined earlier in this chapter, we can include the axiom $Child \equiv Person$ as every person in the world is a child of another person. This is equivalent to saying that a child is subsumed by a person, and that a person is also subsumed by a child, meaning all children are persons and all persons are children. The equivalence axiom would be replaced with two subsumption axioms in EL as follows:

$$Child \equiv Person \Leftrightarrow \{Child \sqsubseteq Person, Person \sqsubseteq Child\}$$

Although EL is very limited, an EL ontology called SNOMED CT has been created to model medical terms. Its earliest ancestor is SNOP, created in 1965, and SNOMED currently contains over 350,000 concepts and nearly 400,000 axioms [11]. It is widely used in the medical industry as it is the largest repository of medical terms [23], and is constantly updated with new releases twice a year.

However, when creating an ontology and saving it to a file on a hard drive, an ontology designer does not use set notations and symbols we used so far in this thesis. Different formats and conventions can be used, and one of them is OWL, short for Web Ontology Language. Even within OWL there are different versions and formats, and we use the functional syntax of the second version of OWL in our work. Created by the Worldwide Web Consortium (W3C) and based on XML, OWL has allowed us to manipulate readable documents and was a practical choice.

The objective of description languages, including EL, is not only to represent knowledge, but to also allow software to understand data and deduce new information based on what is known. This is called *reasoning* and it could follow different methodologies, all having common assumptions. We describe them next.

2.4 Reasoning Approaches and Assumptions

When an ontology has been defined, some rules can be applied on the data set to determine the *satisfiability* of concepts and the concept hierarchy. The process of finding all subclasses in an ontology is defined as *classification* [12]. We will limit our discussion in this section to classifications in EL. As for satisfiability, a concept is satisfiable if it can exist in the modeled

world. In other words, a concept is satifiable if it can have at least one instance. It is then subsumed by Top. We represent a satisfiable concept A using the following axiom:

$$A \sqsubseteq \top$$

In contrast, a concept is unsatisfiable if no instance of it can exist in the domain. In a formal axiom, this concept is subsumed by the bottom concept:

$$A \sqsubseteq \bot$$

Ontologies might only allow concept and role definitions, or might also allow the existence of instances, also known as individuals, or instances of concepts. A set of axioms containing only definitions of concepts and roles is called a TBox, short for *terminological box*. On the other hand, a set of axioms composed only of individuals and their membership to different concepts is called an ABox, short for *assertion box*. Individuals in an ABox can be thought of as objects in object-oriented programming, while the concepts are comparable to classes. In a canonical model, such as the one presented in this thesis, we consider each concept as an individual of that concept. Hence, for simplicity, we can represent edges linking concepts, but in fact, they can only link individuals of those concepts, if they exist.

Reasoning on a TBox might have classification and/or satisfiability of concepts as objectives. In addition to these objectives, reasoning on an ABox might also look for assertions. In other words, a reasoner might want to determine for each individual all the concepts it belongs to. When reasoning, a system looks for *entailments*, logical consequences in the ontology, to make them *inferred* knowledge. The inferred knowledge is a set of new axioms that are added to the ontology by the reasoner. We now proceed to describe some of the different reasoning approaches.

1- Abductive reasoning:

   Abductive reasoning is the act of supposing a hypothesis, then looking for a confirmation that it is either true or false in the axioms [15].

2- Inductive reasoning:

   Inductive reasoning first supposes a hypothesis about a concept derived from its individuals, then tests this hypothesis against known instances of the concept. It then

verifies that this hypothesis does not apply to individuals known to not belong to this concept. [17]

3- Deductive reasoning:

As defined in [5], deductive reasoning occurs when the axioms are exhausted in an attempt to infer all possible entailments. They are applied on the ontology until nothing can further be deduced. In the remainder of our discussion, we will be focusing on this type of reasoning.

Regardless of its type, reasoning has a few properties. First, it is *monotonic*. This means that once an axiom has been inferred, it can not be deleted from the ontology. In addition, inferred knowledge must not clash with prior knowledge. A *clash* is defined as the contradiction of axioms. Conversely, a *tautology* is redundant knowledge. For example, supposing that an ABox includes an individual *Mark Twain* which has been found to be of type *Person.* Knowledge inferred in the future can never be that *Mark Twain* is not an individual of type *Person.* By definition, if any concept exists and is negated by the concept *Person*, then *Mark Twain* can never be found to be an instance of it.

Another assumption is that reasoning always occurs in an open world. Supposing that an ontology only contains two disjoint concepts, *Person* and *Animal*. If an individual *Computer Science* is created in the ontology and is not found to belong to *Person,* then we can never infer that *Computer Science* is an animal, nor that *Computer Science* is not a *Person*. The open world assumption guarantees that inferences follow directly from axioms, but not from their absence.

The final important assumption of reasoning is that concepts might have different names. By comparison to programming, a variable *a* is not the same as variable *b*. Each variable has its own copy of the data, and changes applied to one are not automatically applied to the other. Even if one variable points to the other, the pointer would be located at a different memory address than the variable itself. However, in description logics, a concept can have different names. This is in fact the reason why the equivalence constructor exists. As ontologies model our world, we expect synonyms to occur. For instance, the scientific name for the animal family of ants is *Formicidae*. Hence, an ontology of the animal world might have two equivalent concepts, one named *Ant* and another *Formicidae*. Equivalence also extends to roles. For example, the role we previously

introduced *has spouse* can also be named *is married to.* Both nomenclatures are valid, and if they exist in one ontology, these roles should be declared equivalent.

For a reasoner to be considered of acceptable quality, it must satisfy three conditions. We describe them in the next section.

2.5 Reasoner Quality

A decision procedure is a process yielding a correct solution to a decision question [16]. For a reasoner to be considered a decision procedure, it must be *sound*, *complete*, and *terminating*.

A system that terminates is easily defined as a system that reaches a decision and exits without errors [12]. However, a reasoner that takes years to classify an ontology can not be considered as terminating just because it eventually exits. The reasoner must return a result in a reasonable amount of time.

As for *soundness,* a reasoner is considered to be sound if all the knowledge inferred by this reasoner is indeed entailed, or in other words, a logical consequence of the axioms originally used as input [12]. For instance, a system only given the first two axioms below that infers the third is not sound, as this axiom does not logically follow (1) and (2).

$$(1)\ \ Pen \sqsubseteq Object$$
$$(2)\ \ Window \sqsubseteq Object$$
$$(3)\ \ Pen \sqsubseteq Window$$

Finally, a reasoner is complete if it can infer all the knowledge that is entailed from the ontology [12]. For example, let us consider the two axioms below.

$$Tennis\ Player \sqsubseteq Person \sqcap \exists Plays.Tennis$$
$$Tennis\ Champion \sqsubseteq Tennis\ Player \sqcap \exists HasWon.Championship$$

A system that can only infer that a tennis player is a person, and that a tennis champion is a tennis player is incomplete, and has only inferred what we call trivial knowledge. In fact, a complete system must also infer that a tennis champion is a person. In addition, it must create *plays* role relationships between the individuals of type *Tennis player* and those of type *Tennis*. The same

role holds between individuals of type *Tennis champion* and *Tennis*, and a role *has won* exists between instances of *Tennis champion* and the *Championship.*

For a reasoner to be sound, complete, and terminating, it can be guided by *inference rules.* They are a set of guidelines pertaining to a specific description language and they indicate what axioms can be inferred from existing knowledge. We describe the inference rules for the EL vocabulary in the next section, adapted from [2].

2.6 Inference Rules for EL

We introduced in Listing 2.2 the axioms that can be built using EL. Inference rules are applied on these axioms to add knowledge to the domain.

We denote with S(A) the set of subsumers of A, whether A is a concept or a role. We also denote with $\in$ the constructor *exists in*, with $\notin$ the constructor *does not exist in, union* with $\cup$, and finally, assignment with $\leftarrow$.

In addition, we denote with E(C) the set of role edges outgoing from individuals of type concept C, where each edge is defined using a pair [*role, concept*]. For instance, considering the axiom previously defined $Tennis\ Player \sqsubseteq Person \sqcap \exists Plays.Tennis$, we would add to the set E(*Tennis Player*) an edge [*Plays, Tennis*]. We would then write:

$$E(Tennis\ Player) \leftarrow E(Tennis\ Player) \cup \{ [Plays, Tennis]\}$$

We now start by listing the rules, adapted from [2]. In each rule, we denote concepts with C_i and roles with R_i.

Listing 2.3: Inference rules for EL

IR1: **if** ($(C1 \sqcap C2 \sqsubseteq C3)$ **and** $\{C1, C2\} \in S(C_i)$ **and** $\{C3\} \notin S(C_i)$) $\Rightarrow S(C_i) \leftarrow S(C_i) \cup \{C3\}$

IR2: **if** ($(C1 \sqsubseteq C2)$ **and** $\{C1\} \in S(C_i)$ **and** $\{C2\} \notin S(C_i)$) $\Rightarrow S(C_i) \leftarrow S(C_i) \cup \{C2\}$

IR3: **if** (C1 ∈ S(C2) **and** C1 ⊑ ∃R.C3 **and** { [R, C3] } ∉ E(C2)) ⇒ E(C2) ← E(C2) U { [R, C3] }

IR4: **if** ((C1 ⊑ ∃R.C2) **and** { [R, C2] } ∉ E(C1)) ⇒ E(C1) ← E(C1) U { [R, C2] }

IR5: **if** ((R1 ⊑ R2) **and** {R2} ∉ S(R1)) ⇒ S(R1) ← S(R1) U {R2}

IR6: **if** ((R1 ⊑ R2) **and** {R1} ∈ S(R3) **and** R2 ∉ S(R3)) ⇒ S(R3) ← S(R3) U {R2}

IR7: **if** ({[R1, C1]} ∈ E(C2) **and** R1 ⊑ R2 **and** {[R2, C1]} ∉ E(C2))

⇒ E(C2) ← E(C2) + {[R2, C1]}

IR8: **if** ({ [R, C2] } ∈ E(C1) **and** {C3} ∈ S(C2) **and** ∃R.C3 ⊑ C4 **and** {C4} ∉ S(C1))

⇒ S(C1) ← S(C1) U {C4}

Inference rules that are consequences of the above:

IR9: **if** ((∃R.C1 ⊑ C2) **and** { [R, C1] } ∈ E(C_i) **and** {C2} ∉ S(C_i)) ⇒ S(C_i) ← S(C_i) U {C2}

IR10: **if** ((C1 ⊑ C2) **and** {C2} ∉ S(C1)) ⇒ S (C1) ← S (C1) U {C2}

In the above listing, we do not write an inference for the equivalence axiom, as it will eventually be replaced with two symmetric axioms that we can apply IR2 to.

Each of these inference rules is governed by an *if* condition. The condition needs to be TRUE so that a consequence occurs. This consequence is the addition of an axiom to the ontology. For example, S (C1) ← S (C1) U {C3} will add the following axiom to the ontology C1 ⊑ C3.

The last element in each of the *if* statements in Listing 2.3 ensures that no tautology will occur, in other words, that no redundant knowledge will be added to the ontology.

Reasoning can follow different algorithms and methods to apply the same set of inference rules. We now describe how reasoning can take place with concepts and roles modeled as a graph.

2.7 Reasoning as a Graph Problem

One method of reasoning is to manually update the ontology by adding axioms using the symbol notation, and another is to model the problem as a graph then draw and label concepts and roles as needed. As GRUEL treats the problem as if it were a graph problem, we will only describe this method, paired with deductive reasoning, in this section.

Let us first start by designing a traditional nuclear family ontology using the EL vocabulary, where marriage is a must for all parents. For conciseness, we place symmetric axioms resulting from equivalences on the same line. Like the previous family ontology, this one is not complete.

Listing 2.4: Traditional nuclear family

1- (a) Person ⊑ Child (b) Child ⊑ Person
2- (a) Child ⊓ Female ⊑ Daughter (b) Daughter ⊑ Child ⊓ Female
3- (a) Child ⊓ Male ⊑ Son (b) Son ⊑ Child ⊓ Male
4- (a) Parent ⊓ Female ⊑ Mother (b) Mother ⊑ Parent ⊓ Female
5- (a) Parent ⊓ Male ⊑ Father (b) Father ⊑ Parent ⊓ Male
6- (a) ∃hasChild.Person ⊑ Parent (b) Parent ⊑ ∃hasChild.Person
7- (a) ∃hasParent.Person ⊑ Child (b) Child ⊑ ∃hasParent.Person
8- (a) ∃HasSpouse.Person ⊑ Spouse (b) Spouse ⊑ ∃HasSpouse.Person
9- Spouse ⊑ Person
10- Parent ⊑Person
11- hasDaughter ⊑ hasChild
12- hasSon ⊑ hasChild
13- hasChild ⊑ hasSpouse

Before we start reasoning, we draw an initial graph to represent the ontology. This graph will contain three parts. First, we separate the *taxonomies* from the graph where reasoning will occur.

These taxonomies are tree structures representing the hierarchies for roles and concepts. Usually, a concept taxonomy will include the Top and the Bottom concepts and each unsatisfiable concept would be placed below the Bottom concept. Then, every concept found to be satisfiable would be drawn below Top in the hierarchy. However, as we are reasoning in EL, then by design, all concepts are satisfiable, so we will omit both Top and Bottom from our taxonomies. Figure 2.1 shows the initial state of our ontology graph.

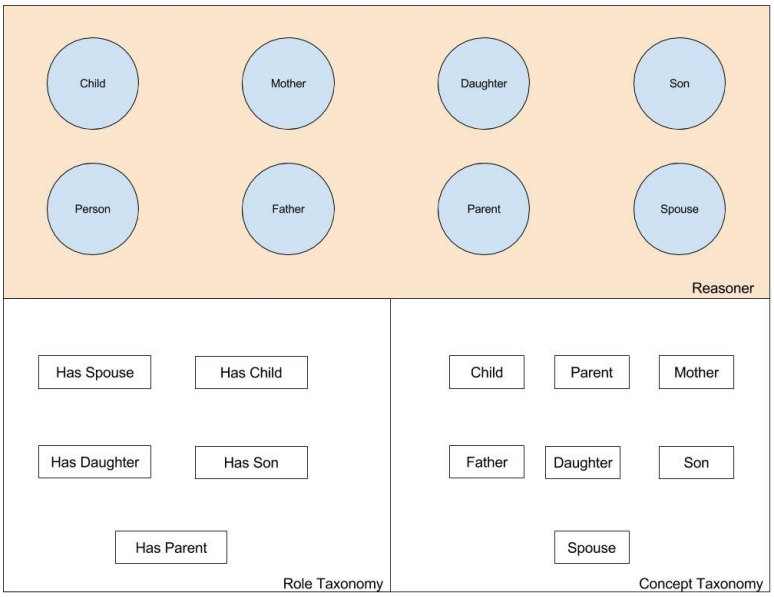

Figure 2.1 Initial ontology graph

In Figure 2.1, we draw each concept as a node and prepare the labels for the taxonomies. No hierarchy is yet found for roles or concepts, and the reasoner has not yet started to make deductions about the axioms. We now begin by applying axioms, starting with 1 (a), 1 (b) and merge the Child and Person nodes on the reasoner side and in the concept taxonomy. Due to axioms 11, 12, 13, the role taxonomy gets updated. Back to the concept taxonomy, applying 2 (a) and 2 (b), Son and Daughter are placed under Child. Applying 9, Spouse becomes a sibling node of Son and Daughter. Because of 4 (b) and 5 (b), a second tree hierarchy is created with Parent at the root and with Father and Mother as leaves. However, using axiom 10, this root becomes a child of Person. The reasoner nodes now get updated with the labels of all ancestors of each node. We use a bold font for the most specific concept name for each node. For Child and Person, as these concepts are equivalent

and their nodes have been merged, both labels are bolded. Our graph is now represented in Figure 2.2.

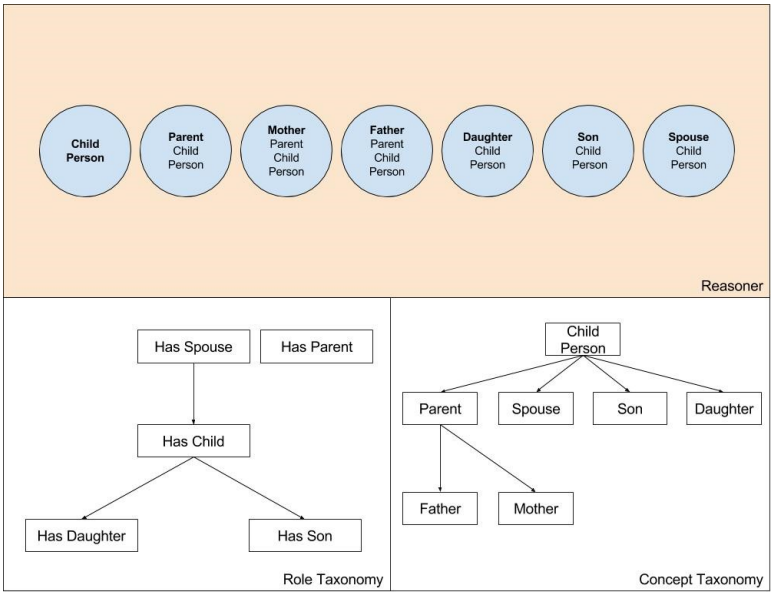

Figure 2.2 Ontology graph after applying all axioms except 6, 7, 8

Applying axiom 7 (b), we create an edge outgoing from Child to Parent with a label *hasParent.* However, since all concepts are subsumed by Child, we create an outgoing edge with this label from each node. To keep our graph clear, we label only the destination side of the edge, and draw the *hasParent* edges in black.

Applying 6 (b), we create an edge from Parent to Child labeled *hasChild.* But according to our role taxonomy, we find that *hasChild* is subsumed by *hasSpouse*. Hence, the edge now has two labels. Due to axiom 8 (a), Parent now becomes a Spouse. This information is reflected on the Parent node label in reasoner, and in the concept taxonomy. In this node, the most specific classification is Parent, so its font remains bolded. As this is a subsumption and not an equivalence, the Spouse node is not affected by this new discovery. Next, because Mother and Father are subsumed by Parent, then the same rules apply, and they each become Spouse with *hasChild* and *hasSpouse* outgoing edges. We represent these edges in purple to avoid duplicating the labels, and color the label Spouse in dark blue.

Applying 8 (b) again on Spouse, we need to create an edge between Spouse and Person labelled *hasSpouse*. We use dark blue for this edge as well. The resulting graph is in the next figure.

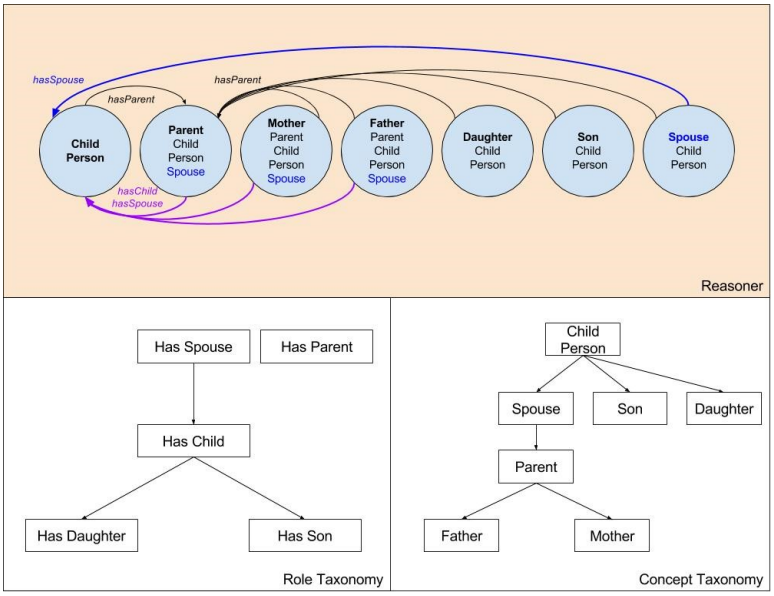

Figure 2.3 Updated graph after applying axioms 6, 7, 8

One discussion that we have delayed so far concerns the Female and Male concepts. There are different ways to represent gender in an ontology, and we choose the simplest, although not the most effective method in this example. We now add both concepts to the concept taxonomy. Doing so, Mother and Daughter will be subsumed by Female, and Father and Son will be subsumed by Male. This means that Mother will now be labelled with Parent, Child, Person, Spouse. Due to axiom 2 (a), Mother will now become a Daughter, and similarly, Father will become a Son. Had we introduced the definition that a female spouse is a wife, then Mother would have become a wife too. There are endless possibilities to expand this ontology, but given only the axioms in Listing 2.4, Figure 2.4 shows the complete classification and taxonomies. For clarity, we use *CP* to denote the Child and Person label, and color in green the Female, Male, Daughter, and Son labels that we have just added to the nodes.

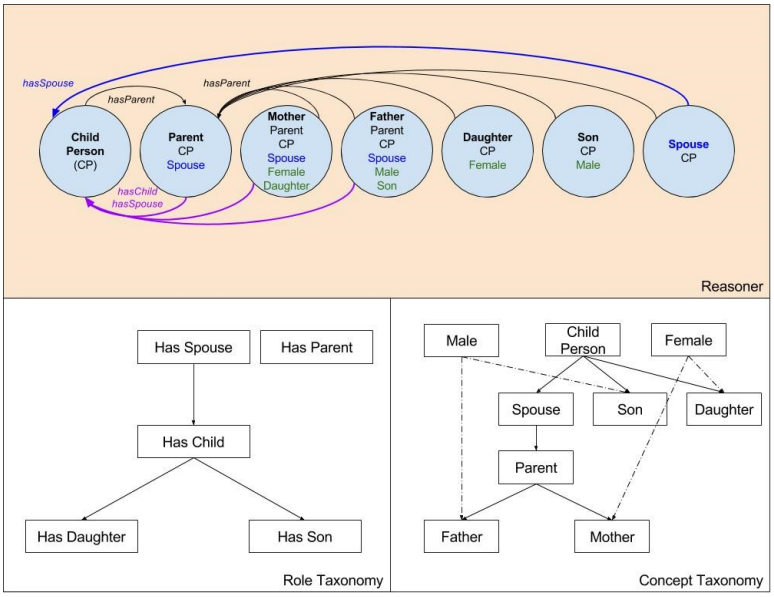

Figure 2.4 Complete classification and taxonomies for ontology in Listing 2.4

One of the advantages of graph reasoning on ontologies is the ease of representation in a program, where adjacency matrices or lists can be used. How we represent the data in GRUEL will be explained in detail in Chapter 5, and we now continue our discussion with the motivation behind the research in knowledge representation and description logic.

2.8 Motivation for Knowledge Representation

As mentioned in the introduction, knowledge representation and logic-based reasoning initially requires input from humans and provides results with 100% accuracy. What seems like a lot of work has many advantages and applications, making the results worth the effort. We have already introduced SNOMED CT, the ontology for medical terms, used by doctors and healthcare professionals to track patient history and standardize medical terminology.

Another application is in the field of software engineering called Lassie created by AT & T and was presented in [7] in 1991. The objective of Lassie was to reduce the amount of time software engineers spent searching for sections of code or for variables, as the code that needed maintaining contained over a million rows. An ontology was created to facilitate the search, where instead of

simple string matching searches, programmers could search specifically for variables, limit their searches to certain files, or use any other property of a certain variable to find it in the code [3].

The application we would like to focus most on is the semantic web, an expansion of the worldwide web. It uses knowledge representation and description logics to map hierarchies and relationships between terms we use in our daily life, and between pages, articles, user data, and other public data on the web. For the semantic web to function, content on the worldwide web needs to be properly tagged. This means that a significant amount of metadata should exist for all data, that this metadata should follow standards and perhaps most importantly, the metadata must use uniform terminology. The backend of the semantic web would consist of large ontologies and fast agents capable of searching them to answer queries.

The goal behind the semantic web is to provide the users of the worldwide web with a tool to access all their information by promoting data sharing between applications and then combining that data using the relationships defined in ontologies [22].

The semantic web would allow for more intelligent search engines by "understanding" exactly what the user is looking for. For instance, let us imagine that a device equipped with a web browser is shared between two people. One of them is a child and a fan of cartoons, especially Mickey Mouse and his best friend Pluto. The other is a student in middle school with an assignment on the solar system, and has chosen the dwarf planet Pluto as a subject for his paper. The younger person's frequent searches and positive feedback on cartoons tagged with or titled "Pluto the dog" will bias a probabilistic search engine towards favoring cartoons as a search result. When the older person will search for "Pluto", most if not all the results on the first page will be of Disney's Pluto.

Refining the search terms to "Pluto in space" might return images of Pluto the dog dressed as an astronaut. Supposing that after the student was able to find the results needed for the assignment, the younger person now wants to search for Pluto, Mickey Mouse's friend. Now, searching for only the keyword "Pluto" would lead to results of the planet, as the search engine would follow principles somewhat similar to temporal locality in memory management. At this point, searching for "Pluto not the planet" would result in articles about Pluto being a dwarf planet.

This is just one scenario where statistical inferences such as those used by Bayesian systems might fail. If these events occur on a semantic web search engine, then "Pluto in space" would have been

understood to have different meanings and we would find among the top search results the images of Pluto in the astronaut suit, in addition to articles about Pluto the planet. This example is not to say that probabilistic reasoners are not good enough, but to simply show that certain use cases exist where logic-based reasoners perform better, especially at quickly providing users with tailored results. That is in fact one of the reasons why Google has started building its *knowledge graph* a few years ago and began using it to answer search queries since 2012 in its *Hummingbird* search engine update.

However, one of the impediments to the quick rise of the semantic web is the fact that content currently on the internet is not uniformly designed or tagged. Serious efforts must be exercised to standardize metadata of webpages and content so that worldwide web users can truly benefit from the strengths and advantages of the semantic web.

2.9 Summary

Knowledge representation is a discipline aiming to model different parts of our world. To achieve this modeling, special-purpose languages are used. Coupled with inference rules, represented knowledge can lead to new information about the domain in what is called logic-based reasoning. A multitude of languages exist with different levels of expressivity, and the choice of language for an ontology depends on its use. One of the main applications of description logics is the semantic web, an extension of the current version of the worldwide web promising to map data with relationships and to provide users with a more tailored one-stop-shop solution to their use of the data they either own or wish to find online.

GRUEL, our proposed solution for a reasoner for the EL language, executes on a graphical processing unit (GPU). As programming for this hardware is different from programming on a CPU, we describe in the next chapter the specificities of "general purpose computing on a GPU".

3. General Purpose Computing on a GPU

3.1 Introduction

Graphical Processing Units have drastically evolved over the past few decades. Currently, a single GPU can have up to thousands of cores, and multiple Gigabytes of local memory. Although the instruction set on a GPU is smaller than the instruction set on a CPU, and although each individual GPU core is slower than a single CPU core, the mere number of cores installed on a single chip and the level of parallelization they provide has made them appealing to programmers, especially when working on compute-intensive applications. As the chips were used to create programs aiming to solve non-graphical problems, the term "General Purpose computing on a Graphical Processing Unit", GPGPU for short, was coined. In this chapter, we provide a summary on designing and implementing a parallel program with GPUs to solve general problems, before proceeding to describe GRUEL.

3.2 Architecture of a GPU

A typical GPU normally consists of a large number of Arithmetic and Logical Units with a few, small memory storages, contrary to a CPU where there are relatively few ALUs and large memory storage caches. For instance, in early 2017, Intel® has released the i7-7920 HQ processor. It has 4 cores and 8 MB of cache memory, yet can support up to 64 GB of random access memory. It can also operate with up to 8 threads [13]. On the other hand, the NVIDIA ® Geforce GTX 1080 released in mid-2016 has 2560 cores and 8 GB of standard memory configuration [18]. Both these models can be installed on a regular desktop. CPUs and GPUs have memory buses to allow data transfers to occur between one chip and the other. In 2011, AMD ® announced the first chip to include both a CPU and GPU with a shared memory, which reduced the memory limitations for the GPU, as well as the data transfer time between both sets of processors, but at the expense of functionality for both units [4].

The device that we have used for the development of GRUEL is a NVIDIA ® Quadro K620. It has 384 cores, with 2 GB of double data rate type three synchronous dynamic random access memory (DDR3 SDRAM). It is not an integrated chip and we needed to explicitly perform data transfers to and from the device's local memory. This chip follows the CUDA ® architecture, which we discuss next.

CUDA is short for "Compute Unified Device Architecture". It is an architecture designed by NVIDIA to make their GPUs programmable for general purpose. Amongst the differences between CUDA and non-CUDA generations of GPU, CUDA devices allow the processors random access to the device's shared memory, and the on-chip ALUs are accessible by non-graphical user programs. In addition, they have a larger instruction set and follow IEEE requirements for floating-point calculations [19]. NVIDIA GPUs with CUDA can be programmed using the CUDA C language, built based on C.

After presenting the architecture of a GPU, we now proceed to explore some of its limitations.

3.3 Limitations of a GPU

First, the memory on a GPU is relatively limited. This means that we can not transfer any amount of data to it. In addition, we need to carefully choose the algorithms that manipulate the data on the GPU, as there might not be enough storage for intermediate results. In our solution, these limitations have impacted our decision regarding the data structures, where we opted for simplification in the representation of our data, picking short integers and Boolean variables wherever this was possible. In addition, we replaced strings names for our variables with integer unique identification numbers and used maps to seamlessly work with either identifier. The decision to replace strings with integer unique identifiers also guaranteed faster performance.

Another limitation was the inability of a GPU to manipulate dynamic variables. For instance, as we could not predict in advance the number of variables our solution needed to work with, we had to first create dynamic data structures such as array lists and vectors, then evaluate their respective sizes, create the matching static data structures and copy the data to the latter. As we mapped the concepts and roles as graphs, we used adjacency matrices. However, working with matrices on the

GPU required a level of indirection, so we replaced all multi-dimensional arrays with one-dimensional arrays, and used a flattening algorithm.

Finally, different sections of code and different variables had to be declared and compiled depending on where they will be used. For instance, a variable and a method created and used on the CPU had to be declared as such, whereas methods and variables created and used on the GPU side also had to be declared and compiled on the GPU. If any element of our solution was to be used by both the host (CPU) and the device (GPU), then a compiler directive indicating this flexibility was to be used. This meant that calling methods and accessing variables from one chip to the other was not straight forward, and specific measures needed to be taken. We explore such steps in the next section, by describing how a GPGPU program flows.

3.4 Flow of Program on a GPU

Using the NVIDIA® Nsight compiler, a program starts by running a kernel file, which includes the main method. This section of the code is run by the CPU. Then, variables need to be copied to the GPU for the device to access them. This copy operation starts by creating pointers for them on the device, then allocating the appropriate amount of memory for each variable. It is only after the copy instruction has been executed that the GPU can read and write these variables. After the GPU operations are completed, then a copy in the inverse direction needs to be performed, if the data is to be accessed by the CPU. Below is a diagram showing how the control flows in a CUDA application.

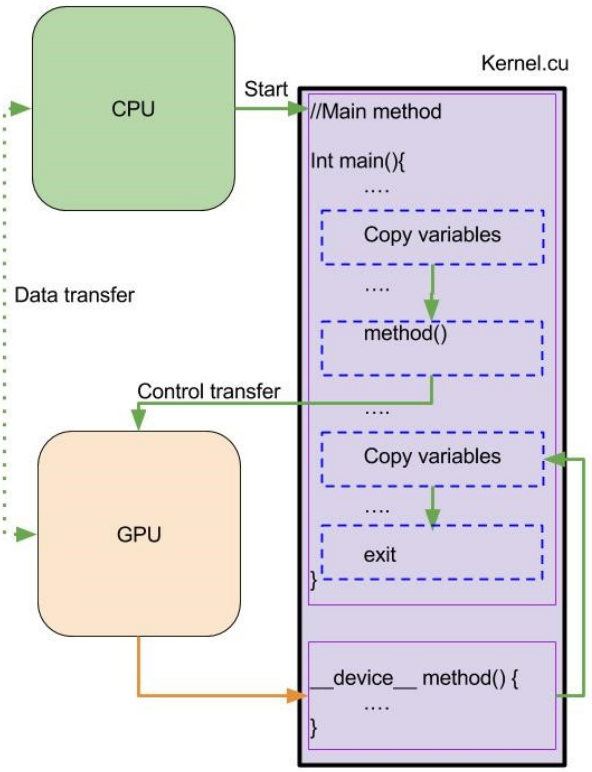

Figure 3.1 Flow of control in a CUDA application

The program starts when the CPU calls the main method in *kernel.cu*. After some preliminary operations, the variables that the GPU needs to work on should be copied to the device. This copy operation is coded into the main method or any other host or global method called by the main method, and is performed by the CPU. The direction of the copy is from the CPU to the GPU. Then, the method compiled on the device using the "__device__" compiler directive can be executed. If need be, this method can call other device methods, or global methods. Global methods are functions compiled to run either on the CPU or on the GPU. When a device method is called, the GPU threads are created and launched, and the application becomes multithreaded. Each thread will execute the code in the device method defined in *kernel.cu*. After this method exits, the control is automatically transferred back to the CPU, and the GPU threads are terminated. If needed, another copy operation can be performed to transfer data back from the GPU's local memory to the system's main memory so that additional operations can be performed on it. The application is now a single threaded application again, and this second copy is also performed by the CPU. After some pre-exit operations, the program can finish and terminate. In the next section,

we explain the two different methods for sharing data between the CPU and GPU that we used when developing GRUEL.

3.5 Memory Access

As previously mentioned, sharing data between the host and the device is not straight forward. A variable needs to be declared as a device variable using a compiler directive, and then copied back and forth between CPU and GPU as needed. Alternatively, it can be created on the CPU, then a pointer is created to designate it on the GPU. Finally, space on the GPU must be allocated to hold the variable, and that is when the copy operation from CPU to GPU can be performed.

To copy data in the opposite direction, from GPU to CPU, the host pointer is used as a final destination for the data. Ideally, all copy operations are wrapped by a method to handle errors if any should occur. In the next diagram, we show the states of the CPU and GPU memory at each of the stage of a copy operation from CPU to GPU.

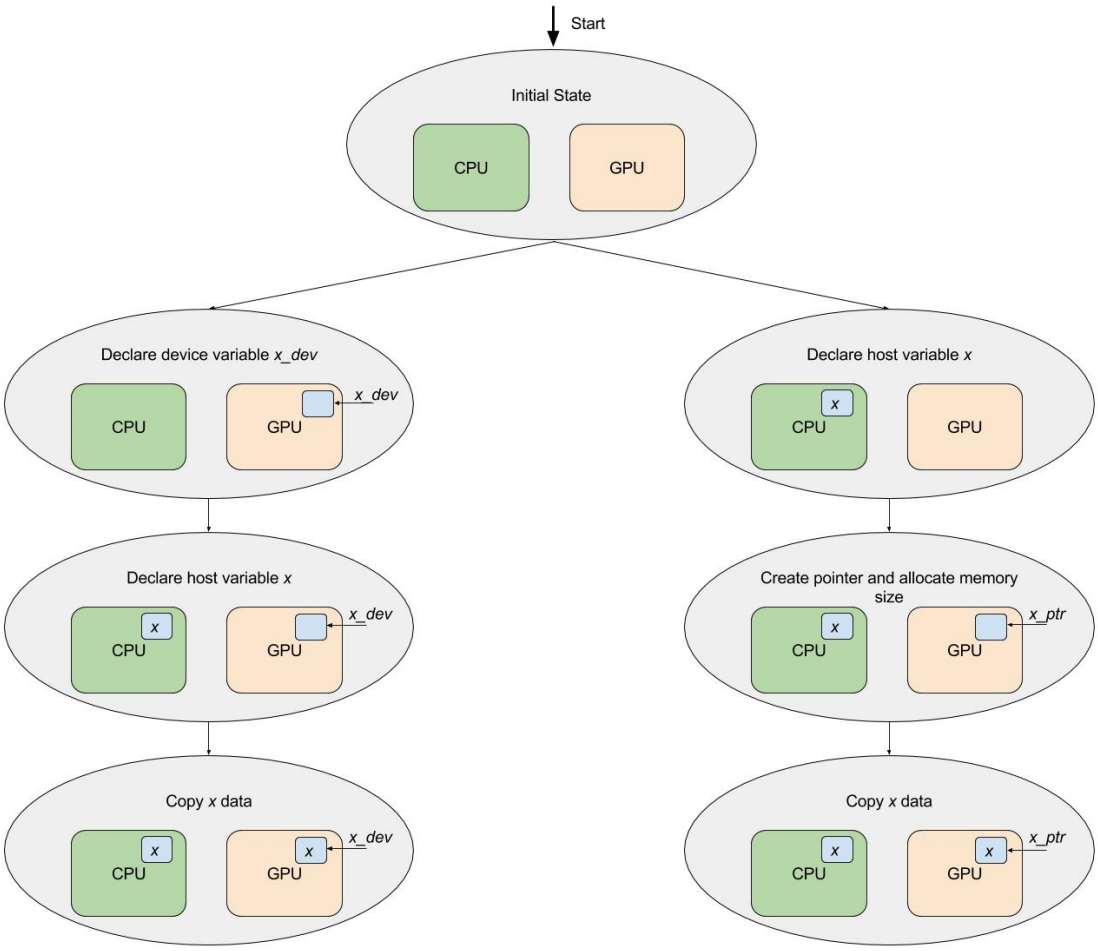

Figure 3.2 Memory state transition diagram

After a successful copy operation, the GPU is ready for the threads to be created, launched, and to start manipulating the data.

Data can be kept on the GPU in different locations. At the lowest level, each core has a set of registers. Then, the design offers a block of random access memory shared by all processors in what is called a *streaming multiprocessor*. A streaming multiprocessor is a group of processors cooperating on the same task. Each streaming multiprocessor has its own shared memory. The highest level of memory on the chip is the cache memory, accessible and shared by all streaming multiprocessors [4].

Outside the chip, a GPU also provides different memory structures. First, the global memory is the memory that a programmer can copy data to and from a program running on the CPU. Then,

constant memory is used for read-only data. Finally, texture memory provides additional storage and is managed by hardware. The different parts in this memory hierarchy are linked by buses. The next figure is annotated with typical bandwidth and latency for each bus [4].

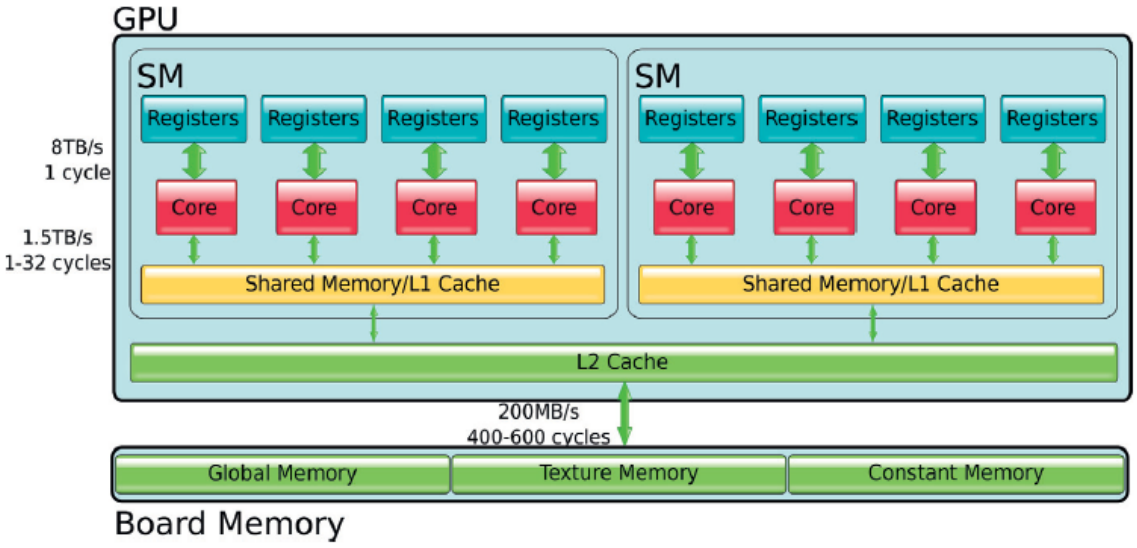

Figure 3.3 GPU memory hierarchy. Taken from [4]

Now that the GPU memory hierarchy is covered, we discuss in the in the next section general threads, GPU threads, and GPU thread partitioning.

3.6 GPU Threads

A thread is comparable to a small process, sharing code and certain data with other threads belonging to the same process. Working with multiple threads saves memory space and reduces context switching time on a central processing unit. In addition, depending on the processor's thread model, a single processor can run multiple threads concurrently, reducing the execution time of the application [20].

As for GPU threads, they are threads created and executed on GPU cores. The CUDA architecture divides the threads into a six-dimensional structure by default, but the programmer can choose to

work with less dimensions. At the highest level, the threads are separated into grids. These grids are structures of blocks of threads, which can be one, two, or three dimensional, where each thread block knows its own location on the grid's axes. For instance, if a three-dimensional grid is used, then each block of threads is assigned an ID on each of the *x, y,* and *z* axis.

Each element of these grids is a block of threads, as mentioned. A block is a grouping of threads which can also have up to three dimensions. However, older versions of CUDA can only accommodate single and two-dimensional blocks. In a block, each thread knows its location, and is also assigned an ID on each axis used. Thus, every thread can be assigned a task, and its location can be used to decompose the data [4]. Figure 3.4 taken from [4] shows the GPU thread hierarchy.

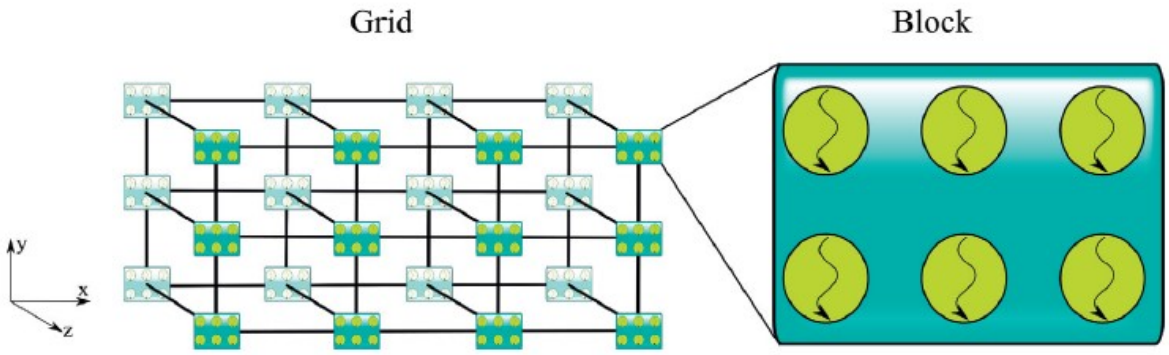

Figure 3.4. GPU thread hierarchy [4]

In Figure 3.1, we show the kernel calling the device method in pseudocode notation. However, when writing the program, it is this call to the device method that contains the block and grid sizes in each dimension. In the CUDA model, a block is further divided into multiple warps. One thread from each warp is assigned to one core. Warps are also groups of threads; their maximum number is limited by the hardware. On the device we used for developing GRUEL, the maximum warp size is 32 threads. To execute a kernel on a CUDA device, the programmer first picks the dimensions of the grids and the blocks. Then, the device separates the blocks into warps. While the warps execute concurrently, blocks execute synchronously. Each thread in a warp has its own set of registers on a core, thus there is no context switching overhead when switching execution from a warp to another. Instructions for a warp are fetched in batch in order to minimize the

memory transfer overhead as well. Thus, CUDA programs have the best performance when the entire warp executes the same set of code. Otherwise, when branching occurs, parts of a warp might be stalled until the instructions can be fetched, making their assigned cores idle, thus reducing the efficiency of the application and amortizing the speedup brought by parallelization. [4]

In multi-threaded applications, special caution must be exercised when sharing data to always guarantee its consistency. In the next section, we discuss the different methods used to ensure data consistency.

3.7 Synchronization

As introduced in [20], when multiple threads cooperate to solve a single program on shared data, a programmer must carefully synchronize the threads, especially the sections of code accessing the data. These sections of code are called “critical sections” and the situation where multiple threads access the same data, and where the integrity and state of the data depends on the order of execution of the threads is called a “race condition”. The objective of using synchronization is to avoid race conditions.

Some methods of synchronization consist of blocking access to the shared data to all threads but one, until this thread has finished updating the data. This can happen using either hardware or software constructs, and monitors, semaphores, or synchronization hardware can be used when a multithreaded application is running on a CPU.

On a GPU, a programmer can use blocking calls, events, or other built-in synchronization constructs. In our solution, we only use the barrier synchronization method in CUDA C, and we will now explore how the barrier synchronization works, and the situation it is useful in.

As the name suggests, the barrier synchronization creates a barrier to stop all threads from executing any further statements at a certain point in the code. The barrier is only lifted when all the active threads in an application reach it. At this point, they can all proceed. Let us suppose that a multi-threaded application draws an image then prints it. If each thread draws one single pixel, the command to print the image should not be issued to the printer before all the threads have

finished drawing. Otherwise, the image printed would be incomplete. In such a situation, a barrier command would be in the code between the drawing method and the printing method. Thus, it is only after the image is complete that the print job begins. A similar situation occurs in our solution as we will discuss in the next chapter, and we use the barrier synchronization to avoid race conditions and the transfer of incomplete data.

In the next section, we explore parallel programming and its specificities.

3.8 Parallel Programming

Parallel programming is defined as the programming which executes multiple tasks simultaneously. This type of programming is different than programming sequential applications and is typically challenging to learn and switch to from the traditional sequential programming [4]. It requires thinking and designing in a different paradigm. It is also more prone to errors, especially those related to synchronization.

Micheal J. Flynn came up with a taxonomy in 1972 classifying computing machines into four categories [9].

First, the *Single Instruction Multiple Data* model. This model is used when the same operation is performed on different data [10]. For example, let us suppose that a program needs to sum the elements of two equally-sized integer arrays and store the result in a third array. The following equation governs the computation:

$$Result\ [i] = Operand1\ [i] + Operand2\ [i]$$

The summation of each element is independent from the others, and this computation qualifies to be parallel-programmed using the SIMD architecture. We can instruct different cores to concurrently execute the summation at different locations of the arrays. We would be thus executing the same instruction on multiple data.

The second model is the *Multiple Instruction, Multiple Data* architecture. In this case, there are different instructions executing on different data [10]. We can think of multiprocessor computers as MIMD machines, with different cores executing different programs on different data and in

different memory locations. Parallel machines can use this model, as multiple tasks can be executed concurrently, especially if the data is independent.

Thirdly, the *Multiple Instruction Single Data* model, which can also be used to design a parallel machine or software. In this architecture, the tasks are repeatedly executed on the same data set. It is most useful when precision is critical, such as in military applications [4].

Finally, the *Single Instruction Single Data* architecture is the model when a machine only has one processor installed, and this processor can only run one program at a time [4]. Older models of computers are SISD machines, and parallelization can not take place. Further additions and ramifications to the taxonomy have taken place since its inception.

For instance, the CUDA model follows an SIMT architecture: *Single Instruction Multiple Threads.* When a block is dispatched to a CUDA machine, there will be multiple threads executing concurrently. As previously mentioned, the instructions will be fetched in a batch and executed on different parts of the data, depending on the location (ID on grid, block, and warp) of the physical core, justifying the name of the classification [4].

Like sequential programming, parallel programming must start with the design of an algorithm. To design the solution, a programmer or software designer minds multiple considerations and makes choices leading to a pattern of decomposition of the problem. Amongst these considerations, one must determine which sections of the solution can be parallelized. Next, the programmer must decide how to allocate a processor to each of the concurrent tasks. The data of the program must then be partitioned, and the access to it managed. Finally, synchronization is considered for the solution. At this point, we can determine which decomposition pattern is best to use given the problem, and the parallel algorithm is finalized [10].

3.9 Decomposition Patterns

Decomposition patterns fall into multiple categories, all with the objective of determining a pattern for the splitting of the parallel tasks. First, decomposition can happen either for tasks, for data, or for data flow. We call these decompositions *functional*, *domain,* or *data flow* respectively. Each of these decompositions can then be performed linearly or recursively, except for data flow

decomposition, which can be either regular or irregular. For instance, regular data flow decomposition leads to a pipeline design. A functional recursive decomposition leads to a divide and conquer approach, while a linear functional decomposition is called *task parallelism*. The decomposition that fits our problem the most was the linear domain decomposition, also called the *geometric decomposition* [4].

Going back to the array addition problem described in Section 3.8, a geometric decomposition would be applied, as it is the best fit for problems where data is structured in arrays or matrices, and where processors can be assigned different data at locations in the matrix split based on one, two, or more dimensions. Whenever the data is independent, such as in this example, the problem is described as *embarrassingly parallel*, as concurrent execution is evident, easy to implement, and provides the highest speedup due to no dependency between the data, and by consequence, there is no need to block threads for synchronization purposes [4].

A pattern may have either a *Global Parallel, Locally Sequential* general program structure, or be *Globally Sequential, Locally Parallel.* The difference between both is the location and frequency of occurrence of tasks that can be parallelized in the software. The single instruction, multiple data model in Flynn's taxonomy, and by consequence, the geometric decomposition and GRUEL fall under the globally parallel, locally sequential category [4]. The contrast between both categories is further illustrated below in Figure 3.5.

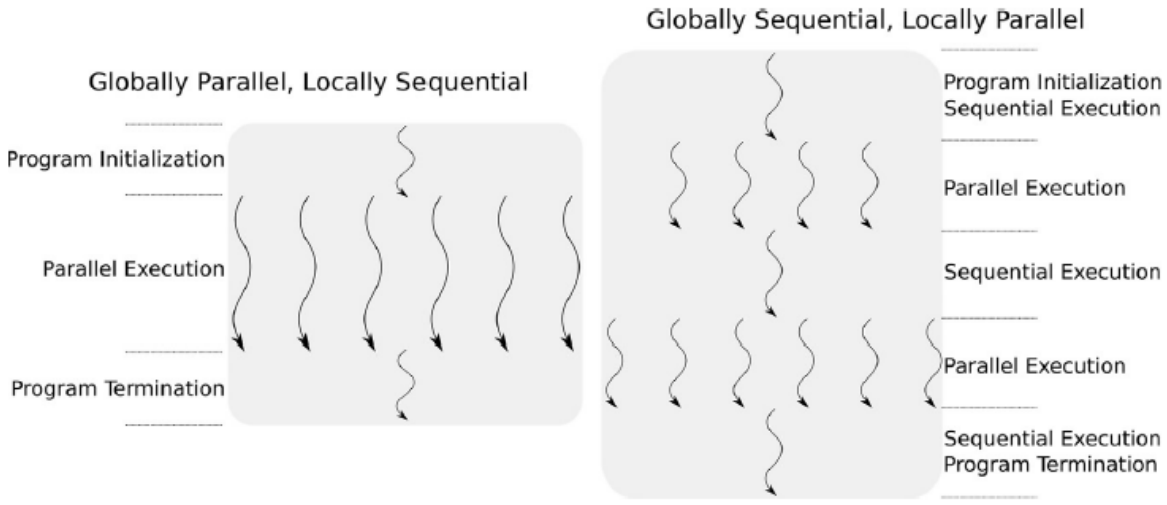

Figure 3.5 Difference between GPLS and GSLP [4]

3.10 Summary

Writing a parallel solution is not intuitive, especially when programming on a GPU. Special considerations must be taken into account when designing and programming the solutions, including and not limited to the memory available, the latency of data transfers, synchronization, decomposition patterns, and program structures.

In the next chapter, we discuss GRUEL and how it was implemented.

4. Architecture Of GRUEL

4.1 Introduction

We implemented an EL reasoner using general purpose computing on a GPU with the CUDA C programming language. GRUEL is thus a "GPGPU Reasoner Used for EL". As we were designing it, we made some algorithm design decisions, in addition to data structure choices to overcome the limitations of the device, all the while attempting to make the solution as efficient as possible. We also needed to keep in mind that the code will be run by parallel cores, and we needed to consider synchronization as well as different methods for decomposing the problem into smaller subparts. In this chapter, we lay down the architecture of the solution and the algorithms it contains. We also describe some of our data structures and justify our choices.

4.2 The Main Algorithm (GRUEL)

We begin by exploring the general structure of the program, described in the below flowchart.

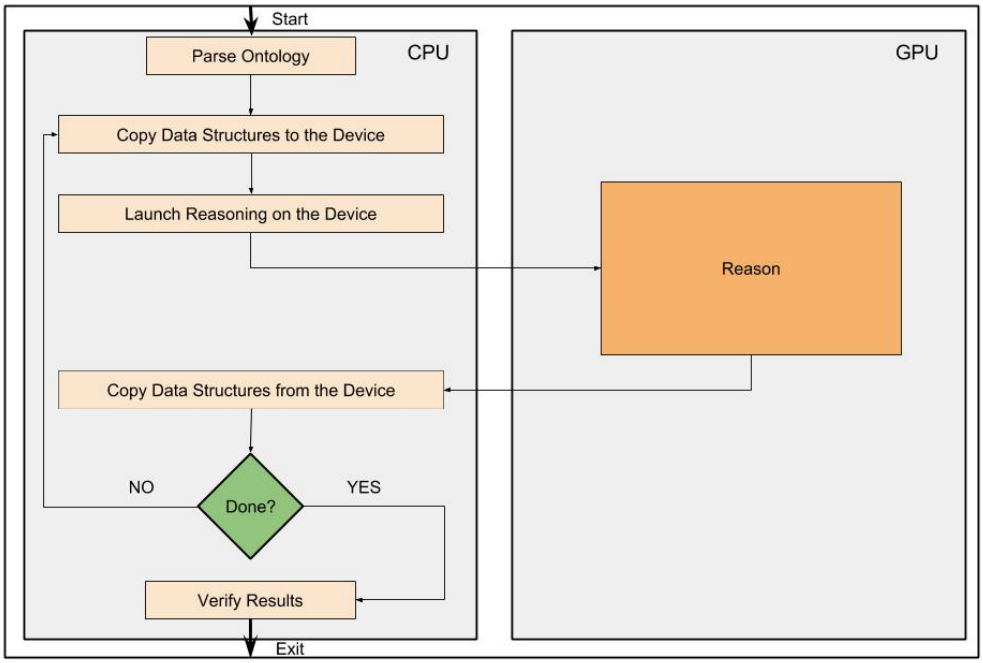

Figure 4.1 Global flowchart for the solution

The program first starts by parsing the ontology and extracting information about the concepts, object properties, and axioms. Once these data structures are saved, they are copied to the GPU for the reasoning to begin. New knowledge may or may not be inferred, depending on the data, and that is when the data structures are copied back to the CPU. A decision is made whether the reasoning is completed, and if it is, we verify the results of our classification by comparing it to the results of a classification by another reasoner called HermiT. If, on the other hand, the reasoning was not yet completed, we re-iterate over the copying and reasoning until no further knowledge can be inferred, and then proceed to the verification.

The algorithm governing these steps is shown next.

Algorithm 4.1: Reasoner (file: *ontology*, file: *verification*)

data structures ← Parse ontology (*ontology*)
while (new knowledge can be inferred **or** this is the first iteration)
do
    copy *data structures* from the CPU to the GPU
    launch reasoning on the GPU
    copy *data structures* from the GPU to the CPU
output (verify *data structures* against *verification file*)

The Reasoner algorithm expects two input files in *.owl or *.txt format, both following the OWL 2.0 functional syntax standard. One of them is the ontology to be classified, and the other is the result of a classification by a different reasoner used for verification purposes. In our experiments, we have chosen to use Protégé, an ontology reader and builder with its HermiT reasoner plugin.

Protégé is designed and implemented by Stanford University and allows multiple reasoner plugins from different developers. One of the reasoners available is HermiT, developed by researchers at University of Oxford. The versions we worked with are Protégé 5.0.0, RC2 build, and HermiT version 1.3.8.

The most important data structures created and set by the parser are the following:

1- The concepts:

   We use a map to assign an integer ID for each concept name found in the ontology. The map is also used throughout the program, to either lookup the name of the concept using its ID, or look up the ID using its name.

2- The roles:

   Similarly, we create a map to hold the role names and their integer ID's and perform lookups using either key.

3- The axioms:

   A vector of integers holding information about each axiom. Ideally, if this were a two-dimensional array, the number of rows in this matrix would be equal to the number of axioms in the ontology, with a constant column count of 5. In the first column, an axiom ID is assigned. In the second, the axiom type is saved. The third, fourth, and fifth columns are used to save the ID's of the concepts and/or roles participating in the axiom. However, as the GPU works easily with simple, static data structures, this vector is one-dimensional, where matrix rows are stored one after the other. For instance, the 5^{th} axiom will occupy the cells 25 till 29 in the vector.

The parser also creates and sets a second set of variables after having completed reading the ontology. Conceptually, these data structures should be multi-dimensional arrays. However, to be able to use them on the GPU, we had to transform all matrices into one-dimensional arrays, and calculate row and column numbers based on cell number using the division and modulo operators respectively. For the remainder of this description, we will assume that a 2-D or 3-D structure is used, and refer to the cells using the bracket notation " [ ] ", as the details of the flattening operation and those of the calculations of cell location are not relevant to the work achieved.

1- The concept subsumption matrix:

   A Boolean matrix of size $n * n$ given n concepts and functions like an adjacency matrix between concepts. For a cell in *[i] [j]* in this matrix, the Boolean value determines whether

a subsumption exists between concept *i* and concept *j*. In other words, given two concepts with ID's *i* and *j* respectively:

$$Concept\ subsumption\ matrix\ [i]\ [j] = TRUE \quad \Leftrightarrow \quad i \sqsubseteq j$$

2- The role subsumption matrix:

Like concept subsumption matrix, the role subsumption matrix is a Boolean matrix of size $m * m$ given *m* roles, where the following holds for two roles with ID's *k* and *l*:

$$Role\ subsumption\ matrix\ [k]\ [l] = TRUE \quad \Leftrightarrow \quad k \sqsubseteq l$$

3- The role edge matrix:

A three-dimensional Boolean matrix of size $n * n * m$ where we assign a value of TRUE to a cell whenever an edge exists between two concepts. For instance, to signal the existence of an edge outgoing from concept *i* to concept *j* where the role is *k*, the role subsumption matrix cell at location *[i] [j] [k]* will be equal to TRUE. In other words:

$$(AXIOM\ TYPE\ 3) \quad i \sqsubseteq \exists k.j \quad \Leftrightarrow \quad Role\ edge\ matrix\ [i]\ [j]\ [k] = TRUE$$

Also,

$$(AXIOM\ TYPE\ 2) \quad \exists k.j \sqsubseteq b \quad AND\ Role\ edge\ matrix\ [i]\ [j]\ [k] = TRUE \quad \Leftrightarrow i \sqsubseteq b$$

Considering the example above, for an axiom of type 2, we need to find if an incoming edge of type *k* exists to concept *j* to determine whether the subsumption holds. We look for the edge by looping around the cells of the role edge matrix assigned for incoming edges to *j*, verifying if an edge matching role *k* exists**.** If one is found, then the concept at the source of this edge is subsumed by concept *b*.

Hence, whenever an axiom of type 3 is encountered, a Boolean value will be changed in the role edge matrix, whereas if an axiom of type 2 is encountered, one row of size *n* is checked, and a value may or may not be changed in the concept subsumption matrix. The role edge matrix is also involved in other types of axioms, as we will explain shortly.

In this design, we have used adjacency matrices to represent subsumptions and role edges between concepts. The concept subsumption matrix and role subsumption matrix are a direct application of adjacency matrices for unweighted directed graphs. The role edge matrix is a slight variation from the same model. Although using adjacency lists would have probably resulted in a lower space utilization, it would have also required more complex tasks and by consequence longer execution times on the GPU. That is why we chose this design, opting for Boolean values to counter the effect of the many elements that we expect in each matrix.

As for the general execution of the program, we use different sub-algorithms mentioned in Figure 4.1. Each of them is described briefly below and then discussed in further detail in the remainder of this chapter.

1- Parse ontology:

   Reads the original ontology and extracts information about concept names, role names, and axioms. It then creates and fills with default values the data structures needed for reasoning, namely, the axioms, the role edge matrix, and both subsumption matrices.

2- Copy data structures to the device:

   Copies the data structures required for reasoning from the CPU (host) to the GPU (device). Although this section of the program was not difficult to design or to implement, it is a crucial step as was explained in Chapter 4.

3- Launch reasoning on the device:

   This algorithm starts by determining how many axioms each core will execute, then transfers the control of the execution from the CPU to the GPU. On each core, it also determines the ID of the axiom to be executed next, and then calls the Reason algorithm.

4- Reason:

This algorithm is the core of the reasoning, where each axiom is executed, potentially incurring changes on all the matrices except the one holding the axioms, which remains unchanged throughout the execution. The algorithm is executed at least once, and keeps executing until no new information can be inferred from the axioms. The iteration is implemented using a while loop with a Boolean flag condition.

5- Copy data structures from the device:

To execute the axioms on the most updated data structures on the device, we must copy the matrices to the CPU, then back to the GPU, except for the axioms. For this reason, a section of the code is dedicated to this copy operation between the GPU to the host CPU.

6- Verify results:

The program proceeds by using two algorithms. One to create an output file and write all subsumptions to it, and the other to read and parse the classification by HermiT, then compare the subsumption matrix resulting from the verification file to the one resulting from the program. The output file was useful for text comparison at the time of testing and debugging, as it is easily readable by humans, as opposed to the matrices which are faster to work with as a machine, but harder to view and evaluate by us.

4.3 Development Environment

The solution was developed using NVIDIA's Nsight Eclipse Edition version 7.5, on 64-bit Linux Centos 7 with CUDA C 7.5.

The Nsight Eclipse Edition is a compiler by NVIDIA, the manufacturer of the GPU installed on the machine we used. Some debugging was performed using NVIDIA's Nsight Microsoft Visual Studio edition due to limitations on the Eclipse edition.

4.4 Iteration and Termination

As mentioned in the main algorithm section, a Boolean flag is used to decide whether an additional iteration of the *reason* algorithm should be performed. If new information is inferred, this flag's value will be set to TRUE, and another execution will occur. The loop is broken once a call to the *reason* algorithm causes no change to any of the matrices. We will now provide an example to prove why such a mechanism is needed.

Suppose the original ontology contains three concepts, A, B, and C, and the following axioms:

$$A \sqsubseteq B \qquad (1)$$

$$A \sqcap B \sqsubseteq C \qquad (2)$$

In the first execution of the *reason* algorithm, axiom (1) will set two Boolean values to TRUE. First, a cell in the concept subsumption matrix to denote the subsumption between A and B, and then the flag, called *changeFlag*. However, at the time of starting this execution, the Boolean value in the concept subsumption matrix was FALSE. Thus, as axiom (2) is being evaluated simultaneously with axiom (1), the left-hand subsumption is not yet discovered, and no concept in the ontology is found to satisfy axiom (2). Consequently, axiom (2) will not infer any new information.

As *changeFlag* is set to true, *reason* will be executed again. *changeFlag* is first reset to FALSE, and all matrices get updated on the device. Evaluating both axioms, axiom (1) will not cause any changes, as the corresponding Boolean has already been set, but the condition for axiom (2) is now satisfied, as concept A is both A and B. The following subsumption will now be represented in the matrix, and *changeFlag* will once again be set to TRUE.

$$A \sqsubseteq C$$

Because *changeFlag* has been set again, *reason* will be called once more. As no new information is discovered, *changeFlag* remains equal to FALSE at the end of the execution, and this terminates the loop.

The procedure of the program is shown in the next table.

Time	Preconditions	Resulting Inferences	Flag value
Start	None	None	FALSE
After Iteration 1	None	$A \sqsubseteq B$	TRUE
After Iteration 2	$A \sqsubseteq B$ $A \sqcap B \sqsubseteq C$	$A \sqsubseteq C$	TRUE
After Iteration 3	None	None	FALSE
Termination	Not applicable	Not applicable	Not applicable

Table 4.1 Sample procedure of program with flag value

We now begin by discussing the details of each major part of our solution, starting with the ontology parser.

4.5 Parsing the Ontology

The program begins by reading and parsing an ontology. The algorithm starts by recognizing the declarative statements, extracts the role and concept names from them, and saves the names with ID's into corresponding maps. Below are sample declarations for a concept and a role.

Declaration(Class(<http://purl.obolibrary.org/obo/UBERON_0006554>))
Declaration(ObjectProperty(<http://purl.obolibrary.org/obo/BFO_00000050>))

The algorithm then looks for equivalence class axioms, and replaces them with two symmetric subclass axioms. It also looks for equivalent role axioms and replaces them with two symmetric sub-role axioms. Next, it searches for class subsumption axioms using a lookup method for the string "SubClassOf", determines the axiom type based on the structure of the line, extracts the participating entities and saves their ID's in the axioms matrix. Finally, it looks for the string "SubObjectProperty" and extracts the information, saving it in the axioms matrix.

The algorithm also makes use of a decision tree when determining the axiom type for concept subclasses, first checking for the most frequent axiom type, (type 1), and then for the others. This is done using a combination of string match searches and line structure patterns. The algorithm is shown below.

Algorithm 4.2: Read ontology (file *ontology*)

```
//extract and store concepts and roles
For each Declaration d in ontology
    If (d is a concept declaration)
        Add the concept name and its ID to the map of concepts
    Else
        //d declares an object property
        Add the role name and its ID to the map of roles

//extract and replace equivalence axioms
For each equivalence axiom in ontology
    Append to ontology two counterpart subclass axioms

//Extract and save axiom information
For each axiom in ontology
    Determine axiom type
    Extract concept and role names
    Find concept and role ID's
    Save axiom information using ID's in the axioms matrix
```

Extraction of names is done by finding special characters and truncating spaces, if any. For instance, below is a declaration of a concept:

"Declaration(Class(<http://purl.obolibrary.org/obo/CHEBI_15377>))"

Finding the name is done by truncating the substring between the last occurring forward slash, and the first occurring closing angle bracket on a line. In some ontologies, the presence of spaces was causing lookup errors in the concept and role dictionaries, so each substring will have its trailing and leading spaces truncated, if they exist.

For space and time efficiency, reasoning on concepts and roles using integer ID's was favored to reasoning using the actual string names. For this reason, the axioms are saved in a vector of short integers, and the maps are implemented using hashmaps of strings and short integers.

We now proceed to the section of the solution where we create the needed data structures for reasoning.

4.6 Creating the Reasoning Data Structures

The GPU has limitations regarding the data structures it can manipulate. For instance, not only is space limited on the device, but it also can not manipulate dynamic structures. As an example, vectors can not be used, but arrays can. In addition, two dimensional arrays are problematic, and using them requires a level of indirection with pointers, so it was decided to work with "flattened" one dimensional arrays instead of matrices. Thus, when reading and parsing the ontology, we save the data into dynamic data structures, then copy them into static structures, after determining their size. We have also decided to work with Boolean variables to save space.

By default, all cells in the Boolean matrices are initialized to FALSE to denote that no relationship exists between any concept and role. The only exception is in the concept subsumption matrix, where each concept is shown as subsumed by itself. The *axioms* matrix is directly filled with the valid axiom information. Figure 4.2 shows how a 7 * 7 concept subsumption matrix looks once it has been initialized with default values.

1	0	0	0	0	0	0
0	1	0	0	0	0	0
0	0	1	0	0	0	0
0	0	0	1	0	0	0
0	0	0	0	1	0	0
0	0	0	0	0	1	0
0	0	0	0	0	0	1

Figure 4.2 Default concept subsumption matrix

We now discuss the next section of the program which copies the data structures from the CPU to the device.

4.7 Copying the Data Structures to the Device

Proceeding with the sequence of the solution, we must now copy the matrices to the GPU. Using the method *cudaMemcpy()*, we copy each *matrix* to the device. Although this section of code is not complex, it is an integral part of the solution, as the GPU has no access to the main memory of the machine used.

For each data structure, we must first create a pointer for it to be used on the device. Next, we need to allocate memory on the GPU to hold the data. Finally, we perform the copy. We wrap these methods in a *HANDLE_ERROR* method, which handles GPU errors, if any occur during these operations.

Below is a code stub showing how we transfer the role subsumption matrix from the system's main memory to the device's main memory. All other data structures are copied similarly.

Code 4.1: Copying data structures to GPU

```
//Create pointer to data exchanged between host and device
bool * roleSubMatrixPtr;
//calculate amount of memory needed
 int spaceForRoles = (rolesCount * rolesCount ) * sizeof(bool);
//Allocate memory on device
 HANDLE_ERROR(cudaMalloc ((void**) &roleSubMatrixPtr, spaceForRoles));
//Copy variable
HANDLE_ERROR(cudaMemcpy (roleSubMatrixPtr, roleSubsumptionMatrix, spaceForRoles,
        cudaMemcpyHostToDevice));
```

If the memory transfer was successful, the program proceeds by launching the axioms on the GPU. We describe this part of the solution next.

4.8 Launching the Axioms on the Device

After having copied the data structures needed for reasoning to the device, we now launch the axioms. In our design, each axiom will be assigned a core to be executed on. There are two cases

to consider. Either the number of available GPU cores is less than or equal to the number of axioms, or the number of cores is greater than the number of axioms.

If we have as many cores as axioms, or more cores than axioms, then the execution is simple. Each core would execute the axiom with an ID equal to its own, and all cores will execute either one axiom or none. However, in the case where there are more axioms than cores, some additional steps need to be taken. We have implemented an algorithm that covers both cases without making this differentiation.

First, in the beginning of each execution, we can read the device properties to find the number of cores it has. As all our tests were performed on the same device, we have skipped this step and hard-coded the number of available GPU cores into GRUEL. Then, we divide the number of axioms by the number of cores, find the ratio *r*, and the remainder *d*. If the integer ratio *r* is 0, then we have strictly less axioms than cores. If the ratio is greater than zero, then there are at least as many axioms as cores. We launch our GPU threads in one grid and one block, so each core will have an ID equal to its position on the chip. All cores will thus have unique identifiers ranging from 0 to the number of cores installed on the devices.

Regardless of the value of the ratio and remainder, the first *d* cores execute $r + 1$ axioms, and the other cores will only execute *r* axioms. Each core will start by executing the axiom having the same ID as itself, then the axiom with ID equal to its own plus the number of cores, iterating either *r*, or $r + 1$ times. The axioms and the cores are both numbered using the zero-based convention, facilitating the implementation of this logic.

Ratio and remainder are calculated after creating the data structures and then copying them to the device, while the looping described happens on the GPU side. After a core exits its loop, we use a barrier synchronization to make sure that all cores have finished reasoning before the control of the application is transferred back to the CPU.

The algorithm governing this operation follows:

Algorithm 4.3: Launch axioms on the GPU cores

```
//part 1: executed on the CPU side, after creating the data structures

Number of cores ← get number of cores on the device
ratio ← number of axioms / number of cores
remainder ← number of axioms % number of cores

//this copy operation happens while copying the Matrices data structures
copy ratio to device
copy remainder to device

//part 2: happens on the GPU, in parallel on each core
if (core ID < remainder)
        //Execute r + 1 times
        For Integer i ←0 till ratio
                axiomID ← self ID + ( i * number of cores)
                Execute axiom with ID equal to axiomID

Else
        //Execute r times
        For Integer i ←0 till (ratio – 1)
                axiomID ← self ID + ( i * number of cores)
                Execute axiom with ID equal to axiomID
Barrier synchronization
```

As mentioned in Chapter 3, GPUs can function in blocks and grids of threads. In our application, we do not expect to need more threads than GPU cores. Consequently, we combine all spawned threads in one block and one grid, with a maximum number of threads equal to the number of cores. An alternative solution would have been to create as many threads as axioms, thus making use of a higher level of parallelization as the number of axioms grows. However, such a solution would have incurred a communication and cooperation overhead between thread blocks.

By launching the axioms in this manner, we have decided to allocate one core and one thread to each axiom. Thus, we have used geometric decomposition by geometrically decomposing the axioms matrix. The program structure pattern is the single program, multiple data in the globally parallel, locally sequential program structure category. Because new inferred knowledge depends on the most recent inferred knowledge, we added the looping mechanism.

Other alternatives of decomposition were considered, such as event-based coordination, where each thread would hold the subsumptions for one concept. Whenever a change occurs on one

concept, then only axioms in which this concept participates would be reconsidered by sending a message to the other cores. Based on our experience, concept subsumption matrices are sparse, so dedicating one core and thread for each concept would have incurred efficiency losses. Another solution was to create a block of threads for each axiom type. The disadvantage in this case is that blocks can not be synchronized with each other, and we need to make sure that threads have finished executing on the data before we copy the matrices back to the CPU. As axioms of different types can be interdependent, this solution was deemed infeasible. Load balancing was also considered when designing the solution. Some axioms are more complicated than others, and require more operations to be performed, and our system is not load-balanced. Some threads may work for a longer time than others to perform their tasks. Given the nature of the problem, we cannot predict or re-distribute bottlenecks of operations. Therefore, we opted for a barrier synchronization, as the load can never be balanced unless all axioms are of the same type, and all concepts have the same depth in the subsumption hierarchy, a condition that is very rarely satisfied in an ontology.

The second part of the *launching* algorithm denoted with "part 2" in Algorithm 4.3 is implemented as part of the *Reason* algorithm. It guarantees that all axioms will be executed exactly once in each call to *Reason.* The actual execution of the axioms is explained in the next section, as part of the *Reason* algorithm.

4.9 The Reason Algorithm

At the core of GRUEL is the *Reason* algorithm. It is the algorithm that executes the axioms. As mentioned in the previous section, one core might execute this algorithm multiple times, up to a maximum of $\frac{numberOfAxioms}{numberOfCores} + 1$. It starts by determining the ID of the axiom to be executed, its type, and then executes different sections of code based on this type.

This is the design of the Axioms array:

Previous axiom	axiomID	axiomType	Role or concept1	Role or concept2	Concept3	*Next axiom*

As this is a one-dimensional array, the complete information of an axiom is saved between cells numbered *5 * axiomID*, and *5 * axiomID + 4* inclusively. These are the cells to be read before any reasoning takes place.

Next, the algorithm will check the type of the axiom in order to execute the proper code. This is done using a switch case statement. In case the axiom type is not identified, then an error is displayed. The flowchart of the algorithm is shown in the next figure, with each box representing a different section of code, where A, B, and C denote concepts, and *j, k l* denote roles. Each thread executes one axiom, and a barrier prevents threads from proceeding further in the code before all threads have finished reasoning on their respective axioms.

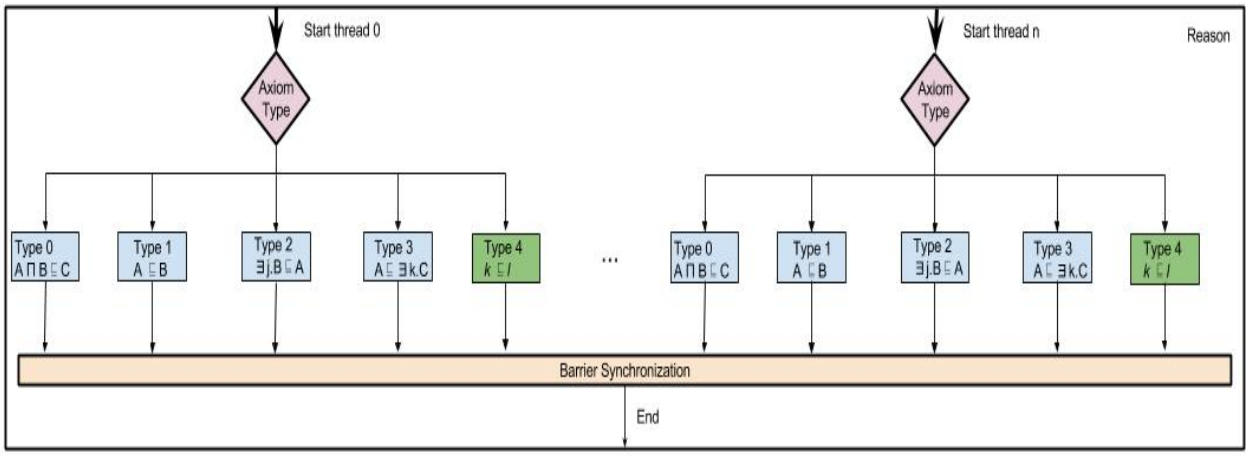

Figure 4.3 The Reason algorithm

Once the data is extracted from the axioms, the condition for subsumption, if any, is checked. If the condition is not satisfied, so for instance, if the axiom is of type 0 and no concept is the intersection of the concepts on the left-hand side of the axiom, then nothing happens. But if the condition holds, the concept at the right-hand side is added as a parent to the concept that satisfied the condition.

To complete the classification, whenever a new subsumer is found, the parents of the new subsumer are assigned as parents of the new subsumee, and the new subsumer is set as a parent to all children of the new subsumee. The Boolean *changeFlag* is then set to TRUE. The completion does not iterate to add all parents of the new subsumer as parents of all children of the new

subsumee because *Reason* will be called again due to the value of *changeFlag*, and all parents will be gradually added to all children, by executing the axioms until no new changes occur. Although this may lead to more iterations, it reduces the number of operations and therefore the execution time of each individual iteration.

Finally, before making any changes to a matrix, we check if the Boolean value has already been set. If it is, then nothing needs to happen. This prevents the program from getting stuck in an infinite loop, continuously setting *changeFlag* and the same Boolean value to TRUE.

Concurrency control did not need to be explicitly handled. As previously mentioned, reasoning is monotonic and the EL expressivity does not contain any negation. Consequently, once a concept or a role has been found to have a child or a parent, this relationship can not be changed or removed. Similarly, once a role edge has been found, it can never be removed, and no concept will ever be found to be unsatisfiable. In addition, all matrices are stored on the device's main memory, accessible by all cores, and none of the data structures is copied to a core's local memory. Thus, all changes to a matrix can only set values from FALSE to TRUE on the shared memory, and no core will overwrite a TRUE value by a FALSE value, race conditions are avoided, and we have the advantage of saving thread cooperation and communication time.

As mentioned in the previous section, the only type of synchronization employed here is a barrier synchronization, to make sure that all threads have finished executing before copying the data structures from the device to the host occurs. A summary of the reason algorithm is now presented.

Algorithm 4.4: Reason (*matrices*)

```
axiomType ← extract axiom type from axioms matrix
case axiomsType of
        value 0: execute algorithm for axioms of type 0
        value 1: execute algorithm for axioms of type 1
        value 2: execute algorithm for axioms of type 2
        value 3: execute algorithm for axioms of type 3
        value 4: execute algorithm for axioms of type 4
        other: output error
endcase
```

Next, we present each algorithm for the different axiom types, starting with axioms of type 0, having the following form, where *C1, C2,* and *C3* are concepts:

$$C1 \sqcap C2 \sqsubseteq C3$$

The algorithm loops over all concepts to check if any is subsumed by both concepts on the left-hand side. As a reminder, a concept is by default set as subsumed by itself in the concept subsumption matrix. If a concept satisfying the condition exists, then it is marked as subsumed by the right-hand side concept. *ChangeFlag* is updated appropriately, and then an algorithm for completion is called. The completion algorithm will be presented in the next sections.

Algorithm 4.4.1: Axioms of type 0 (*matrices)*

```
//axioms of type 0 are of the form c1 ⊓ c2 ⊑ c3 where c1, c2, c3 are concepts

For each concept c in concepts
    if (c is subsumed by (c1 AND c2) AND c is NOT subsumed by c3)
        add c3 as subsumer to c
        changeFlag ← TRUE
        execute completion (c, c3)
```

Axioms of type 1 are the axioms that are most frequently encountered. They have the form $c1 \sqsubseteq c2$ where *c1* and *c2* are concepts. It is not a conditional axiom, and thus, whenever such an axiom is encountered, we add *c2* as a parent to *c1,* if this was not already done, and call the completion algorithm. *ChangeFlag* is also set whenever a change occurs in a matrix, and changes might not occur if *c1* has already been set as subsumed by *c2,* and if the completion steps have been performed with no new changes to the taxonomy of neither *c1* nor *c2.* The completion algorithm is called even if the subsumption in the axiom has already been saved as changes in the set of ancestors for *c2* and in the set of successors for *c1* affect one another.

The pseudocode for reasoning on axioms of type 1 is shown next.

Algorithm 4.4.2: Axioms of type 1 (*matrices)*

//axioms of type 1 are of the form c1 ⊑ c2 where c1, c2 are concepts

If (*c1* is **NOT** subsumed by *c2*)
    Add *c2* as subsumer to *c1*
    ChangeFlag ← **TRUE**
Execute completion (*c1, c2)*

Axioms of type 2 have the form $\exists r1.c1 \sqsubseteq c2$ where *c1, c2* are concepts and *r1* is a role. The algorithm starts by checking the role edge matrix to find concepts having a *r1* edge to *c1*, then if such a concept exists, *c2* is added as its parent and the completion algorithm is called.

Algorithm 4.4.3: Axioms of type 2 (*matrices)*

//form of axioms of type 2: ∃ r1.c1 ⊑ c2, where *c1, c2* are concepts, and *r1* is a role

For each concept *c* **having** an edge of type *r1* to concept *c2*
    If (*c* is not subsumed by *c2*)
        Add *c2* as subsumer of *c*
        ChangeFlag ← **TRUE**
        Execute completion (c, c2)

Axioms of type 3 are structurally the mirror opposites of axioms of type 2: $c1 \sqsubseteq \exists r1.c2$. In this case, we do not need to verify conditions for satisfaction of the axiom, and we should simply add an edge of type *r1* to concept *c2* from concept *c1* and all concepts it subsumes. This type of axiom can only change the role edge matrix. As previously explained in Section 4.2, if an incoming edge from concept *c1* of type *r1* exists to concept *c2,* then the cell in the role edge matrix at location [*c1*] [*c2*] [*r1*] will be equal to TRUE.

Algorithm 4.4.4: Axioms of type 3 (*matrices*)

//an axiom of type 3 has the form: c1 ⊑ ∃ r1.c2, with *c1, c2* concepts, and *r1* is a role

If (role edge from *c1* to *c2* of type *r1* does **NOT** exist)
    role edge matrix [*c1*] [*c2*] [*r1*] ← **TRUE**
    ChangeFlag ← **TRUE**

For each concept *c* **in** *children of c1*
    If (role edge from *c* to *c2* of type *r1* does **NOT** exist)
        role edge matrix [*c1*] [*c2*] [*r1*] ← **TRUE**
        ChangeFlag ← **TRUE**

The last axiom type to cover is of type 4, has the form: $r1 \sqsubseteq r2$ where *r1* and *r2* are roles. Handling these axioms is similar to handling axioms of type 1, and we need to set the subsumption Boolean value in the role subsumption matrix at location [*r1*] [*r2*] and copy all edges of type *r1* to edges of type *r2*. Below is the pseudocode for reasoning on axioms of type 4.

Algorithm 4.4.5: Axioms of type 4 (*matrices*)

//axioms of type 4 are of the form r1 ⊑ r2, with r1, r2 roles

If (role *r1* is **NOT** a child of role *r2*)
    role subsumption matrix [r1] [r2] ← **TRUE**
    ChangeFlag ←**TRUE**

For each edge *e* **in** *role edge matrix*
    If (*e* has role **equal to** *r1*)
        create edge *e'*
        e'.source ← *e.source*
        e'.destination ← *e.destination*
        e'.role ← *e.r2*
        ChangeFlag ← ***TRUE***

Next, we explain the completion algorithm. We define completion as the steps taken to make the subsumption graphs complete. Whenever a new subsumption is found between two concepts, this subsumption will eventually be propagated to all parents and children by repeating the reasoning on the axioms. The subsumed concept and all its children will become children of the new parent, as well as children of all ancestors of this new parent. This is known as maintaining the transitive closure of a graph. According to [14], the transitive closure of a directed graph is another directed graph with an edge between nodes *i* and *j* if and only if *j* is reachable from *i*. Applying this definition to subsumptions, to maintain the transitive closure of our subsumption matrix, then concept subsumption matrix at location *[i][j]* is equal to TRUE if and only if *j* is an ancestor of *i,* regardless of the respective depths of *i* and *j* in the subsumption tree.

However, looping over all concepts, all their children, and all their parents every time a subsumption has been inferred is costly. On the other hand, not performing any completion might be costly in the sense that multiple iterations over the reasoning algorithm will be needed to reach the final state. The trade-off was found by just adding the direct parents of the new subsumer to the set of parents of the new subsumee, and by adding the new subsumer as a parent to the children of the subsumee, thus creating a low-cost shortcut to completion. Below is the algorithm for the completion algorithm.

Algorithm 4.4.6: Completion (concept ID *c1,* concept ID *c2*)

For each concept *c* **in** {children of *c1*}
    if (*c2* is **NOT** a parent of *c)*
        Add *c2* as parent to *c*
        ChangeFlag ← **TRUE**

For each concept *c* **in** {parents of *c2*}
    if (*c1* is **NOT** a child of *c)*
        Add *c1* as child to *c*
        ChangeFlag ← **TRUE**

For each edge *e* **where** the destination is *c1*
    if (edge *e'* = *e* except for destination **AND** *e'.destination* = *c2* does **NOT** exist*)*
        role edge matrix [*e.source*] [*c2*] [*e.role*] ← **TRUE**
        ChangeFlag ← **TRUE**

The completion algorithm has access to the matrices in the system to check for roles, for parents and children. Next, we prove the usefulness of the completion algorithm.

Suppose our ontology contains thousands of concepts where concept Z is subsumed by all concepts except for *A, B, C, D,* and *E*, and where *r* is a role such that:

$$A \sqsubseteq B, \qquad B \sqsubseteq C, \qquad C \sqsubseteq D, \qquad D \sqsubseteq E$$
$$E \sqsubseteq \exists r.Z$$
$$\exists r.Z \sqsubseteq Z$$

To complete the classification, the system would require six iterations to find that $A \sqsubseteq Z$, in addition to the number of iterations needed to find all of *Z*'s subsumers, plus one more, where *ChangeFlag* remains equal to FALSE.

However, using our completion algorithm, the procedure for the program would be as follows. For clarity, we denote with *Z*+ the set of concepts composed of *Z* and all its parents.

Time	Preconditions	Resulting Inferences	Flag value
Start	None	None	FALSE
After Iteration 1	None	$A \sqsubseteq B$ $B \sqsubseteq C$ $C \sqsubseteq D$ $D \sqsubseteq E$ $E \sqsubseteq \exists r.Z$	TRUE
After Iteration 2	$A \sqsubseteq B, B \sqsubseteq C, C \sqsubseteq D \Rightarrow$ $B \sqsubseteq C, C \sqsubseteq D, D \sqsubseteq E \Rightarrow$ $C \sqsubseteq D, D \sqsubseteq E \Rightarrow$ $E \sqsubseteq \exists r.Z, \exists r.Z \sqsubseteq Z \Rightarrow$	$A \sqsubseteq C, A \sqsubseteq D$ $B \sqsubseteq D, B \sqsubseteq E$ $C \sqsubseteq E$ $E \sqsubseteq Z+$	TRUE
After Iteration 3	$A \sqsubseteq D, D \sqsubseteq E, E \sqsubseteq Z+ \Rightarrow$ $B \sqsubseteq E, E \sqsubseteq Z+ \Rightarrow$ $C \sqsubseteq E, E \sqsubseteq Z+ \Rightarrow$ $D \sqsubseteq E, E \sqsubseteq Z+ \Rightarrow$	$A \sqsubseteq E, A \sqsubseteq Z+$ $B \sqsubseteq Z+$ $C \sqsubseteq Z+$ $D \sqsubseteq Z+$	TRUE
After Iteration 4	None	None	FALSE
Termination	Not applicable	Not applicable	Not applicable

Table 4.2 Executing the completion algorithm

Thus, finding all parents of a concept is done in less iterations using the completion algorithm. This algorithm also implements one key inference rule where all role edges with the new subsume as a destination are duplicated and the destination is set as the new subsumer. In case we come across an ontology where there is no indirect parent (or ancestor) to any other concept, and where there are no role edges with subsumed destinations, then the completion algorithm needlessly executes and wastes precious time. However, such ontologies are not very common, and most ontologies contain at least a few subsumption chains, some of them affecting destinations of roles.

After finishing the reasoning, and after verifying that no new knowledge can be inferred, each thread will hit the barrier synchronization to wait for all others to reach the same statement. Then, the control is transferred back to the CPU, and the updated matrices are copied to the host.

4.10 Copying Results from GPU to CPU

After each call to the *Reason* algorithm, the data structures are copied to CPU, then, if need be, are copied back to the GPU for another execution of all axioms. We also copy *changeFlag* to verify its value. The only matrix that does not need to be copied is the one holding the axioms, as they never change. The code is similar to the code used to copy the data from the CPU to the GPU. The only difference lies in the direction of the copy, which is the last argument of the *cudaMemcpy* function. After copying the results, we release the memory on the device. This step would be the equivalent of destroying objects in C++, then calling the garbage collector.

4.11 The Verification

An algorithm is used to verify the results of our classification against the classification produced by HermiT. Using Protégé, we would load the ontology, classify it, then export the inferred axioms into an OWL file using the functional syntax. When such a file is loaded into the system, a matrix similar in design to the concept subsumption matrix is created, and then a cell by cell comparison is performed.

Whenever text comparisons were needed, we would print to a file all the subsumptions inferred by HermiT, then print in the same format all the subsumptions produced by GRUEL, and finally compare them using text comparison tools. In this section, we only discuss the matrix verification algorithm, as the writing algorithm is very simple and straight forward.

Algorithm 4.6: Verify (file *subsumptions by HermiT*)

Boolean *Hermit subsumption matrix* ←read and parse *subsumptions by HermiT*

For each Boolean *b* **in** *Hermit subsumption matrix* at location *l*
    if (*b* **is NOT equal to** *concept sumsumption matrix [l]*)
        return FALSE

return TRUE

Reading the verification ontology proceeds similarly to reading the original ontology file, with two differences. Firstly, we donot read the declarations, and use the maps directly instead. Secondly, as we load the concept subsumption matrix produced by HermiT, we directly ensure transitive closure for all parents and all children of both concepts participating in a subsumption. The second difference exists because when exporting axioms from Protégé, only the direct parents are included in the file. Hence, we must manually discover and map the hierarchy into our matrix. The final step after representing the classification into a subsumption matrix is to compare the Boolean values by location. As we are using the same maps, we can be sure that the same concept will have the same ID in both matrices, and consequently, any subsumption at a certain location in one of the matrices must be present at the same location in the second matrix for the test to pass.

Lastly, we do not verify the role subsumptions as we are more interested in the classification of the concepts. However, the correctness and completeness of the role subsumption matrix affects the classification.

To validate GRUEL, we needed to prove that it meets the reasoner quality criteria introduced in Section 2.5. The inference rules implemented in the solution are a subset of inference rules in [6] and [2]. These papers prove the soundness and completion of these rules and a few others when applied to a vocabulary slightly larger than EL. We simply neglected the rules pertaining to non-EL constructors and implemented the rest, and we can thus safely assume that termination, soundness, and completion are maintained. It is worth mentioning that we did not implement rules that are consequences of others, and let the completion algorithm and the iterations to lead to the same knowledge those rules would have inferred.

4.12 Summary

GRUEL consists of an ontology parser, reading the names of all entities used and creating unique short integer identifiers for them. It then proceeds by creating and initializing data structures, and copying them from the CPU to the GPU, where the reasoning occurs. Each axiom is evaluated repeatedly, until no new knowledge can be inferred, and that is when we verify our results by comparing them to results of a classification by another reasoner called HermiT. The program

makes use of multiple algorithms and mostly employs matrices of either Boolean or short integers for efficiency purposes.

Now that we have described the design of the solution, we proceed in the next chapter to analyze its performance.

5. Performance And Limitations

5.1 Introduction

In the previous chapter, we have described the design of the solution that we propose for an EL reasoner using general purpose programming on a GPU. In this chapter, we will be measuring the performance of GRUEL using various ontologies and comparing it to the performance of other reasoners publicly available. We will then discuss the limitations of the system and the difficulties faced when developing the solution.

5.2 Ontologies Used For Benchmarking

During our testing and benchmarking phase, we used ontologies that we developed from scratch ourselves, as well as ontologies from the Oxford University ontology repository, and from the 2014 OWL Reasoner Evaluation workshop (ORE 2014). In total, we tested GRUEL on around 15 ontologies. All these ontologies are developed using only the EL language. All the ontologies used for testing contain subclass and declarative axioms. In the next table, we show the properties of some ontologies used for testing, in ascending order of the axiom count. Due to the long name of some ontologies, we have given each an alias.

Ontology Alias	Source	Axiom Count	Concept Count	Role Count
577	Oxford University	31	23	2
8633	ORE 2014	629	600	2
1983	ORE 2014	732	453	1
8698	ORE 2014	876	813	3
609	ORE 2014	1058	834	1
3458	ORE 2014	1099	963	6
444	Oxford University	13,730	13,737	2

Table 5.1 Properties of some test ontologies

The actual names of the ontologies used are as follows:

Ontology Alias	Source	ID in original repository
577	Oxford University	00577
8633	ORE 2014	58296df4-ebd3-44ac-bb20-f4dfa2434c8f_pato.owl_functional.owl
1983	ORE 2014	approximated_218b2b45-850e-4905-95fe-c72b97c2f5c4_tinstances.owl_functional.owl
8698	ORE 2014	15642674-7893-4b0c-806b-8b7358526824_nbosimple.owl_functional.owl
609	ORE 2014	approximated_06a23d92-adfc-4e2d-a7e3-92ea3096f84d_pathway_functional.owl
3458	ORE 2014	1eea77e0-2f62-4db9-907e-293b3e8f5852_classified.obo_functional.owl
444	Oxford University	00444

Table 5.2 Original ID of test ontologies

In the next section, we report on the performance of the system when classifying the above ontologies.

5.3 Performance and Comparison to Other Reasoners

We now present the performance of GRUEL in comparison to other reasoners available as plugins to Protégé 5.0. To compare performances, we loaded each ontology separately into our system and ran the reasoner while logging the time needed for reading the ontology, parsing it, creating the data structures, reasoning, and returning the full classification. After verifying the results with HermiT and making sure that the classification was complete and sound, we consider that this test has passed and that the classification was successful. All the ontologies classified have passed the verification by comparison to HermiT's results.

All tests were performed on the same machine, equipped with a NVIDIA ® Quadro K620 GPU, an Intel Xeon ® CPU having 8 cores and a clock rate of 3.60 GHZ. The main memory is a DDR3 of size 15.5 GB. The operating system used when performing the tests was Centos Linux 7.

To increase the testing accuracy, all tests were performed with no other active user processes on the machine besides the application used for the test itself. When benchmarking GRUEL, the only user application was the NVDIA Nsight compiler, and when performing the classification with other reasoners, only Protégé was active on the machine. In between tests, we would close the application and open it again to avoid having some cached data for any of the reasoners or ontologies.

As for the reasoners used for comparison, we used HermiT, Pellet, and ELK. HermiT has already been introduced, and Pellet is an open-source OWL 2.0 reasoner developed using Java [21]. We used the 2.2.0 plugin version in Protégé. As for ELK, it is also an OWL 2.0 reasoner built using Java with an objective to accommodate the OWL 2.0 EL expressivity[8]. The version we used is 0.4.3.

Although we performed tests on approximately fifteen ontologies, we only present a few results as they are sufficient to show the performance trend.

Ontology	**Classification time by reasoner (in milliseconds)**			
	GRUEL	**HermiT**	**Pellet**	**ELK**
577	70	86	129	268
8633	110	183	174	377
1983	80	142	135	276
8698	120	187	142	361
609	90	201	157	254
3458	*Unclassifiable*	225	179	326
444	*Unclassifiable*	1497	717	1438

Table 5.3 Performance of different reasoners for ontology classification

If we load each reasoner only once and perform tests on ontologies successively, then all reasoners but ours provide a better performance and the time taken to classify an ontology can be as low as 25% of the time measured and reported in Table 5.3. GRUEL in this case would not always offer the best performance, as it does not make use of memory caching. In addition, our system still contains a few inefficiencies and is not yet scalable, as proven in the last two rows of Table 5.3. Although GRUEL performs better given the test conditions previously described, we are aware of a few inefficiencies in its design leading to scalability problems and slower execution. The reasons behind the inefficiencies in GRUEL and methods we wish to explore in the future to attempt to overcome them will be discussed in the upcoming sections.

On the other hand, we believe that one reason behind the faster performance of our system despite the optimization and tremendous efforts invested in the development of other reasoners is because GRUEL knows exactly what type of axioms to expect. Apart from ELK which is built specifically for EL, the other reasoners might be looking for other axiom types and for additional relationships between concepts and roles, wasting their time and slowing down their execution.

What is also worth mentioning is that some of the reasoners used for comparison perform some pre-processing when reading the ontology. Consequently, the time needed to only generate the classification is in the order of a few milliseconds because much of the reasoning has already been simplified in the pre-processing phase. Also to keep our comparison fair, we account in Table 5.3 for the time used for reading, pre-processing, and logging for all reasoners.

The below graph compares the trend of performance for each reasoner with respect to the number of concepts. We remind our readers that starting 963 concepts, GRUEL can no longer perform the classification, for reasons described next. We also remove the data point for the 444 ontology, as it made the chart unreadable due to scaling.

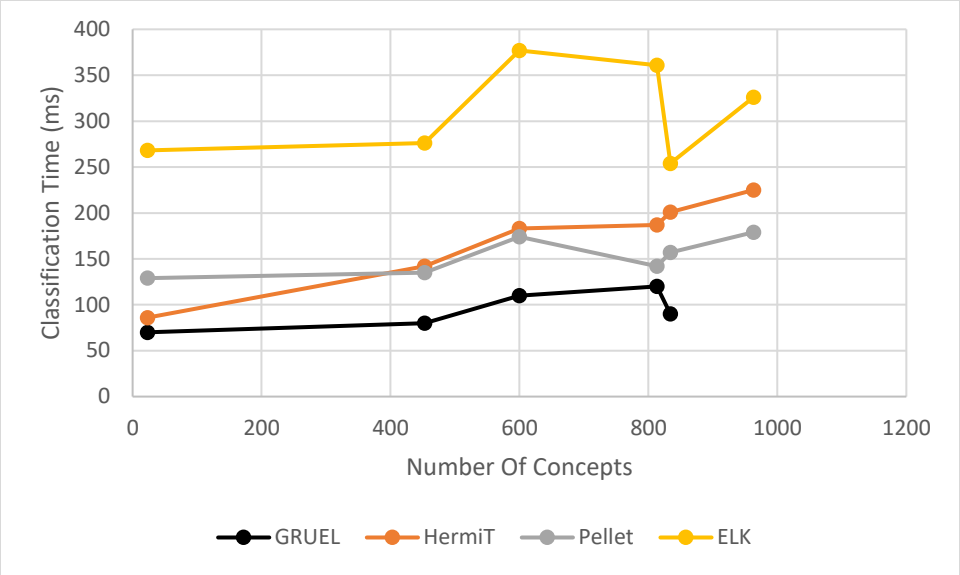

Figure 5.1 Comparison of performance by different reasoners

As space and time efficiency analysis is critical for any programming problem, in the next section, we analyze the complexity of GRUEL, and discuss the reason behind the failure of our system to scale well.

5.4 Space and Time Efficiency

We start this discussion with the space efficiency of the solution, then proceed to an analysis of the time efficiency.

5.4.1 Space Complexity

The major data structures created for reasoning are matrices of either linear or quadratic size. For instance, the size of the axioms matrix is linear with respect to the number of axioms, but the role edge matrix is quadratic with respect to the number of concepts. This has caused one of the limitations of GRUEL, discussed in the following paragraphs. Given n concepts, r roles, and m axioms, the following table shows the space efficiency of each key data structure.

Data Structure	Keeps data for	Asymptotic notation
Concept subsumption matrix	The concept hierarchy	$O(n^2)$
Axioms matrix	The axioms data	O(m)
Role subsumption matrix	The role hierarchy	$O(r^2)$
Role edge matrix	Edges between individuals of concepts	$O(n^2r) = O(n^2)$

Table 5.4 Asymptotic notation of the main data structures

As we expect ontologies to contain many more concepts than roles, the asymptotic notation becomes $O(n^2)$ for the role edge matrix. The quadratic space required for roles edge matrix was the reason why GRUEL could not classify large ontologies. The limit was at around 900 concepts, and we faced a stack overflow error when the system could no longer address all the elements in the array. For 900 concepts, we expect to have two arrays each having at least 360,000 elements. By using a different compiler, namely the Visual Studio with the NVIDIA ® CUDA plugin, we could overcome this limitation, yet faced other issues with the addressing of the data structures in other sections of the code, leading to an even further reduced performance.

We now proceed with the time efficiency analysis.

5.4.2 Time Complexity

As for the time complexity of parallel algorithms, we calculate a few metrics given the formulas in [4]. We use measurements for Pellet and GRUEL for only the 1983 ontology containing 453 concepts, 732 axioms, and one role. The metrics for other ontologies and other reasoners will not be identical, but comparable. We chose Pellet as it is generally faster than the other two reasoners to reduce bias in the calculation. We also chose the 1983 ontology as it has a medium size compared to the other ontologies GRUEL successfully classified. We first start with the *speedup* metric.

The speedup granted by parallelizing a program is expressed as the time needed for a sequential execution divided by the time needed for a parallel execution. The execution times will depend on multiple factors including and not limited to the current state of the machine and any active processes running in the background at the same time as the testing application. Performance is

also affected by the design and implementation of the solution, in turn affected by multiple factors such as the programmer's competency and experience levels [4]. In the case of our tests, one potential source of discrepancy in the speedup is using different applications. GRUEL's performance is measured directly by the program, while the performance given the same ontologies and on the same machine of other reasoners is measured using Protégé. If it exists, we do not expect this potential discrepancy to affect the results, as the operations are timed in the reasoners and displayed in their log, thus bypassing Protégé and avoiding any overheads it could add to the reasoners.

The speedup of GRUEL is:

$$Speedup = \frac{time\ needed_{sequential\ execution}}{time\ needed_{parallel\ execution}} = \frac{135}{80} = 1.6875 \approx 1.69$$

A speedup greater than 1 indicates that the decision to parallelize was a justifiable choice.

Another metric used for time analysis of parallel algorithms is the *efficiency*, defined as the speedup divided by the number of cores available for parallelization. A perfect efficiency would be equal to 100%, as it indicates that all cores execute the same amount of work and that each of them is busy 100% of the execution time. As we already know that GRUEL is not load balanced, we do not expect a high efficiency. As previously discussed, the NVIDIA Quadro K620 chip has 384 cores. We now calculate the efficiency of GRUEL:

$$Efficiency = \frac{time\ needed_{sequential\ execution}}{Number\ of\ Cores \times\ time\ needed_{parallel\ execution}}$$

$$= \frac{135}{384 \times 80} = 0.00439 \approx 0.4\ \%$$

This efficiency is considered low and indicates that the resources are not fully utilized. This means that our design could be improved, and that we should consider investing more time in our solution. As per our findings, we now know that our parallel design and implementation was beneficial but far from optimal.

In non-GPU parallel design, other metrics are also used to evaluate the scalability of systems. However, for programs developed using GPGPU, these metrics become controversial [4] and are

usually avoided. In our case, these metrics will not be used as we already know that scalability is one limitation of GRUEL. In the next section, we discuss other limitations of the reasoner.

5.5 Limitations of the System

Scalability is the biggest limitation of GRUEL. Additionally, some smaller problems also occur within the reasoner. The first limitation pertains to the format of the ontology. GRUEL can only handle correct input for the ontologies files. In case an ontology is not normalized or contains some invalid axioms, the system will display an error message and exit, even if the input error is minimal. It also does not deal with aliases of concepts or roles, contrary to OWL 2.0. In OWL 2.0, any entity can have a unique identifier in addition to an alias, usually more readable than the identifier. However, GRUEL does not account for such declarations of aliases. While testing ontologies that use aliases, we had to manually replace the aliases with their unique identifiers in HermiT's output file for the verification to take place.

The second limitation regards the hard-coding of some variables. In the current version of GRUEL, for instance, we have hard-coded the number of cores on the device. We have also hard-coded the maximum number of concepts allowed to avoid a stack overflow error. Some additional maximum sizes of variables have been hard-coded for the same purposes. Ideally, we should not have such limitations, but if they exist, they should be calculated or read and set at runtime depending on the machine utilized.

An additional limitation that affects performance is the branching of the program. As shown in previous chapters when describing the architecture of the solution, the code branches depending on the axiom type on the GPU. This is not a good practice when using GPGPU as machine instructions are fetched in bulk for each warp. So, for instance, when a warp containing different axiom types is executing, the instructions will be fetched for some of the threads while the others are stalled. We believe this is one of the reasons behind the low efficiency of GRUEL. This limitation is brought by the problem itself where we can not avoid having different axiom types and branching, as well as by our design that indiscriminately launches all axioms in one block.

Finally, parsing the axioms happens by analyzing their structure, and by using string comparisons and manipulations. We consider this a limitation of the system as adding new axioms will require

much effort and will be prone to bugs. An improvement on this method would be a more robust and dynamic determination of the axiom types.

Some of the discussed limitations stem from difficulties faced when developing the solution. In the next section, we describe some of the problems faced when designing and implementing this system.

5.6 Difficulties Faced

Many difficulties arose when working on this system. First, GPGPU was a completely new methodology for us and we had to start by learning it from scratch. Although this difficulty was then overcome thanks to the literature available on the topic and the support of some members of our faculty, our lack of experience in this field forced us to take a trial-and-error approach to the design.

The second difficulty was the limitations of the GPU device itself such as not dealing with dynamic data structures and some complex data types. After finding a potential design, we would start the implementation phase only to find that we are facing an architectural roadblock and had to retrace our steps, re-design our solution, then face another problem and repeat the same procedure. For instance, to overcome the difficulty of working with complex data types on the GPU, we made a design decision to opt for binary adjacency matrices between concepts and roles instead of the more space efficient adjacency lists. However, this resulted in a quadratic space complexity. Coupled with limited memory on the GPU and the stack overflow when attempting to address large arrays, the difficulty was then to optimize GRUEL to allow for a better scalability. We removed some data structures and created some shortcuts in the program which slightly pushed the stack overflow limit, yet this is still not enough.

Third, the NVIDIA NSight compiler occasionally crashed for no apparent reason. Without changing any line of code or modifying the state of the system, a simple restart of the computer would fix the issue and we could proceed with our work. Alternatively, we sometimes needed a series of restart operations to get the application to run again, which seems odd. This was inconsistent behavior though, and we learned to recognize the differences between compiler failure and programming errors, after the first few occurrences when we wasted time attempting to debug

our application when there was nothing wrong with it and the misbehavior came from the compiler itself.

Finally, we could overcome many of the difficulties and produced a system with an acceptable performance although admittedly, still has much room for improvement.

5.7 Summary

In this chapter, we started by introducing the different ontologies and reasoners used for comparing the performance of GRUEL with others. We then proceeded to a space and time efficiency analysis and a discussion of the limitations of the system and some difficulties faced when developing it.

GRUEL performs well yet is not competitively scalable nor efficient in terms of resource utilization. We already have a plan to implement changes and improve this reasoner. These enhancements are to be attempted in the future and we discuss them in the next chapter.

6. Future Work

6.1 Introduction

We have ended our last chapter by discussing the limitations of GRUEL. In this chapter, we describe the planned future work to improve it. For each major limitation, we provide a description of at least one method that we wish to implement, starting with solutions to the most serious limitation, scalability. We then proceed to discuss possible enhancements to the execution time and general expansion plans.

6.2 Scalability

As we have seen in the previous chapter, although GRUEL performs well for small ontologies, it is unable to classify larger ones. To overcome this limitation, we can proceed in two general directions. We can attempt to reduce the space complexity from quadratic to at least linear, or we can address the arrays differently.

To reduce the space complexity of the arrays, we must reconsider our data structure design. At the cost of execution time, we will implement adjacency lists instead of matrices. Because we can not predict the exact number of edges or subsumptions in an ontology, and because a GPU can not deal with dynamic data structures, we will use a work around inspired by dynamic hash tables. In general, in any ontology, we can expect a sparse graph for both subsumptions and edges between individuals of specific concepts.

We can start the application by allocating minimal space on the GPU for storing the adjacency lists and expand them once their occupation rate reaches a threshold. This method would require additional cooperation and communication between the GPU and CPU, as it is the device that will monitor the occupancy while it updates the data structures, but the host CPU is the only one that can modify memory allocations on the GPU. The threshold needs to be below 100% occupancy to avoid losing execution time. For instance, if the threshold is at 100%, when the last element in the list is occupied, all threads that are currently executing will be terminated because they can not

store their results. Setting the threshold slightly lower (at around 75% for example, as is typical for dynamic hash tables) will allow the GPU threads to finish their current iteration while there is likely enough space for all inferred information. If by some chance this iteration happens to add too many elements and the adjacency matrix is full, then the iteration is terminated. Next, the CPU regains control of the application, transfers the data from GPU to CPU for safeguarding, releases the memory on the GPU, then re-allocates larger memory space for the adjacency lists. Finally, the CPU would copy the data back to the device and re-launch the threads.

Choosing the proper threshold, expansion rate, and initial size of the data structures is critical. On one hand, we do not wish to have too many copy operations, and on the other, we still wish to keep an occupancy rate high enough to not waste precious memory space on the GPU with sparse data structures. By analyzing the amortized complexity of dynamic hash tables, it has been proven that the efficiency of this method is highly dependent on the choice of threshold and expansion rate, and to a limited extent, on the initial size too. However, it is known that the space complexity of adjacency lists is $\Theta(V+E)$ where V is the number of nodes and E is the number of edges in a graph. Thus, in the worst case, GRUEL will become linear in space, which already is an improvement to the current design. At the time of implementing this method, we will conduct further analysis and calculations to make the optimal choice, and then we might alter some implementation details after an initial experimentation phase.

Then, we will consider modifying the addressing mode by using indirect addressing for the arrays. Although this might increase the execution time and the complexity of the program especially on the GPU, this method can be worth the effort of implementation and experimentation. One way to indirectly address the arrays would be not to flatten them and keep the matrix design. In CUDA C, because it is built on C++, we expect that multidimensional arrays will be implemented as arrays of arrays and hence, the maximum number of elements allowed before a stack overflow occurs will be squared. An additional "manual" indirect addressing can be coded, where we dynamically split elements of the arrays into different arrays and include a memory translation function that would handle calculating memory addresses and finding the elements in the corresponding sub-array. Of course, we will first attempt to reduce the space complexity of the data structures, then opt for a multidimensional array design. We do not expect to implement the dynamic indirect addressing algorithm unless we plan to experiment with very large ontologies. For the moment,

our target is to classify ontologies of 10,000 concepts, and if this is achieved without a dynamic indirect addressing, then we will not implement it.

After dealing with the limited scalability of GRUEL, we also wish to improve its execution time. The methods we suggest for doing so are exposed next.

6.3 Performance

Although GRUEL classifies small ontologies faster than other reasoners, we are aware of a few inefficiencies in its design. Some improvement can be seen if we tailor the design even further to the CUDA architecture and GPU functionality.

As previously discussed, the GPU fetches machine instructions in batches for each warp. In one warp, threads executing a different code segment than the one that was just fetched are stalled. Warps are not controlled by the programmer, and are handled by hardware. However, each block of threads is divided into warps, and we can control the blocks. Hence, launching axioms of the same type in a single block will guarantee that all warps in that block will be executing the same code. Thus, in one warp, we will achieve better load balancing and consequently an overall better efficiency, as calculated in Chapter 6. One complexity arises here in thread synchronization between blocks. As mentioned earlier, synchronization constructs on a GPU do not work across blocks. In other words, we can synchronize threads in one block, but can not synchronize threads contained in two or more blocks. The advantage of monotonic reasoning is that we only need a barrier synchronization at the end of iterations, and we think that this might be achieved.

Whether on a CPU or a GPU core, the memory transfers are the operations that take the longest to complete. By modifying the blocks of threads that we use in our solution, we hope to reduce the latency and stalling brought on by instruction fetching when the code has branched. The key performance benefit of a GPU is due to fetching instructions while the GPU works on the data. Stalling is thus avoided as much as possible and the memory transfer overhead is eliminated if the cores are all still executing the previous batch of instructions while the next batch is being fetched. Our current design does not fully take advantage of this feature, and this is what we hope to achieve by following this suggested method.

However, the biggest problem with this design is that block size is usually fixed, and we can not guarantee that an ontology will contain the same number of axioms for each type. The solution in this case is to either perform some pre-computations, and/or to split the code into separate fragments to guarantee execution of the same lines of code at the same time on the GPU. We believe that any overhead due to synchronization or additional transfer of control will be counteracted and made up for by the execution time improvement, and a solution with the design described here will provide faster results than the current ones. Only experimentation can prove or deny this claim.

To further improve performance of CUDA C programs, some libraries have been developed. They implement basic algorithms and facilitate working with complex data types and are freely available for programmers. An example of such libraries is the Thrust C++ library built for CUDA. We also plan to explore the functionality of Thrust and how using it can improve GRUEL. We expect that some generic parts of our code have already been implemented in Thrust and we can simply call the methods in Thrust instead of using our own code. We trust that the experience of the developers behind Thrust will reduce the execution time of some of our code fragments.

In addition to performance and scalability improvements, we have high hopes for our system and wish to make it more comprehensive. We now discuss how such efforts will be conducted.

6.4 Going Beyond EL

The EL expressivity is enough for certain applications. As we have seen, SNOMED is currently being used in the medical industry and is developed using only EL. However, description logics contain many additional languages and constructs. Some of them can be added more easily to GRUEL than others. After making it more scalable and after attempting to further reduce its execution time, we wish to expand it to reason on more constructs and axiom types.

For starters, some role properties can be easily implemented in GRUEL. Roles can sometimes be reflexive, which means that for an individual in the ontology, this role applies to itself. For example, narcissistic people love themselves. In an ontology, this is indicated by an individual having a role edge of type *loves* where both the source and the destination of the edge is the same individual. Reflexive properties can easily be included in GRUEL by simply adding a role edge

from a node to itself. We remind our readers that concepts do not have edges but it is only individuals that do. We use a representation that, for simplicity, assumes that in the absence of an ABox, each concept node represents a single individual of that type.

In addition, roles can be transitive. For example, if individual A is parent of individual B, who in turn is a parent of individual C. Then B is an ancestor of C, A is an ancestor of B, and by consequence, A is an ancestor of C. The role *is ancestor* is then transitive, as it is passed on from individual to the other. Subsumptions for instance are transitive, and as the name suggests, transitive roles are too. The design of our system allows for an easy expansion to include transitive roles.

Another expansion regarding roles is allowing role chains. For instance, a person who has a parent who in turn has a sibling, who in turn has a child, is a person who has a cousin. We express such a statement as a role chain, where one role, *has cousin* in this case, is defined using a chain of other roles. A role chain can be as small as two successive roles, or as large as the ontology designer wishes. After having implemented adjacency lists instead of matrices, reasoning on role chains becomes easy as our algorithm will only need to follow the edges, checking the type of the individual at each destination, and then either following that path if it satisfies the role chain, or pruning it. Hence, it is a special case of a path finding algorithm in a directed graph.

Adding the functionality described above to GRUEL allows it to be a reasoner for EL and R+. Role hierarchies, role transitivity and reflexivity, in addition to role chains are part of a description language constructor family denoted with R+.

After successfully building our ELR+ reasoner, we might include even more DL constructors and keep on expanding the system. Although going beyond ELR+ is not in our current plan, we might consider it for the future.

6.5 Summary

In our future work, we will start by making GRUEL more scalable. To do so, we will first redesign our data structures to reduce the quadratic space complexity to at least a linear complexity. Then, if the scalability threshold is still considered to be low, we would consider different addressing

methods for the data structures. Once an acceptable scalability is achieved, we will proceed with software design changes to reduce execution time, and our main objective is to reduce or remove branching on the GPU. Finally, after these improvements, GRUEL will be made compatible with the ELR+ expressivity, and then we might perhaps consider going further in description logic vocabularies.

At this point of the thesis, we have described GRUEL, analyzed it, and provided improvement plans for future work. We now present the final chapter of this document, the conclusion.

7. Conclusion

The objective of our research was to find a method to optimize the performance of a description logics reasoner. That goal was achieved by using general purpose computing on a graphical processing unit, where GRUEL accomplished sound and complete classifications of small ontologies at least 70% faster than other reasoners currently available. The vocabulary we chose to accommodate is limited as we consider this work to be a proof of concept that GPGPU can be used to solve good old fashioned artificial intelligence problems, namely deductive logic-based reasoning problems.

Our work falls in the larger scope of description logics for the semantic web. As some of the world's largest software companies have started using description logics and reasoning, we can only expect that this discipline will gain popularity, and that it will become one of the next global trends.

Until then, we hope that our research is beneficial for both the academic and commercial communities, and are looking forward to witnessing what future will description logics and the semantic web bring us.

Bibliography

[1] Baader, F. et al. 2005. Description Logics as Ontology Languages for the Semantic Web. *Mechanizing Mathematical Reasoning*. (2005), 228–248.

[2] Baader, F. et al. 2005. Pushing the EL envelope. *IJCAI International Joint Conference on Artificial Intelligence* (2005), 364–369.

[3] Baader, F. et al. 2003. The Description Logic Handbook: Theory, Implementation, and Applications. *Description Logic Handbook*. (2003), 622.

[4] Barlas, G. 2015. *Multicore and GPU Programming An Integrated Approach*. Morgan Kaufmann.

[5] Brachman, R. and Levesque, H. 2004. *Knowledge Representation and Reasoning*.

[6] Brandt, S. 2004. Polynomial Time Reasoning in a Description Logic with Existential Restrictions, GCI Axioms, and — What Else? *16th European Conference on Artificial Intelligence (ECAI-2004)*. Dl (2004), 298–302.

[7] Devanbu, P. et al. 1990. LaSSIE: a knowledge-based software information system. *[1990] Proceedings. 12th International Conference on Software Engineering*. (1990), 110–115.

[8] ELK OWL 2 EL Reasoner: *https://www.cs.ox.ac.uk/isg/tools/ELK/*. Accessed: 2017-04-03.

[9] Flynn, M.J. 1972. Some computer organizations and their effectiveness. *IEEE Transactions on Computers*. C-21, 9 (1972), 948–960.

[10] Grama, A. et al. 2003. Introduction to Parallel Computing; 2nd Edition. *Search*. (2003), 856.

[11] Héja, G. et al. 2008. Ontological analysis of SNOMED CT. *BMC medical informatics and decision making*. 8 Suppl 1, (2008), S8.

[12] Hitzler, P. et al. 2009. *Foundations of Semantic Web Technologies*.

[13] Intel i7 Products Index: 2013. *http://www.intel.com/cd/products/services/emea/spa/sitemap/384113.htm?iid=subhdr-ES+products_all*. Accessed: 2017-01-01.

[14] Italiano, G.F. 1986. Amortized efficiency of a path retrieval data structure. *Theoretical Computer Science*. 48, C (1986), 273–281.

[15] Klarman, S. et al. 2011. ABox abduction in the description logic ALC. *Journal of Automated Reasoning*. 46, 1 (2011), 43–80.

[16] Kroening, D. and Strichman, O. 2008. *Decision Procedures*.

[17] Lehmann, J. and Hitzler, P. 2010. Concept learning in description logics using refinement operators. *Machine Learning*. 78, 1–2 (2010), 203–250.

[18] NVIDIA GTX 10-series products page: *https://www.nvidia.com/en-us/geforce/products/10series/laptops/*.

[19] Sanders, J. and Kandrot, E. 2010. *CUDA By Example - An Introduction to General-Purpose GPU Programming*. Addison-Wesley.

[20] Silberschatz, A. et al. 2005. *Operating System Concepts*.

[21] Sirin, E. et al. 2007. Pellet: A practical OWL-DL reasoner. *Web Semantics*. 5, 2 (2007), 51–53.

[22] W3C Semantic Web Activity: *www.w3.org/2001/sw*. Accessed: 2017-03-24.

[23] What is SNOMED CT: *www.snomed.org/snomed-ct/what-is-snomed-ct*. Accessed: 2017-03-22.

[24] 2009. Nuclear Family. *Encyclopaedia Britannica*.