POLITECNICO DI TORINO

Master Degree in
Computer Science - Software Engineering

Master Degree's Thesis

Solving the maximum common subgraph problem on many-cores architectures

A massively parallel implementation of the McSplit algorithm

Supervisor
Dr. Stefano Quer
Associate Professor

Candidate
Gabriele Mosca
Matr. 243909

A.A. 2018/2019

Abstract

The Maximum Common Subgraph (MCS) Problem is a well known problem in literature. It has been proved to be an NP-hard problem, then it is often considered as not worthy to search for an exact solution due to its high complexity. Luckily, the advent of newest computing technologies and processors, several algorithms have been developed to efficiently solve very complex problems. Given their still high time consumption, this thesis discusses the possibility to produce a GPU implementation under the CUDA environment of one of the most efficient strategies currently known, namely the McSplit algorithm Despite we find that such implementation it is indeed possible, experimental results showed that, considering the actual state of the CUDA APIs, it may be at least questionable to use this version against a standard CPU-only multi-thread implementation.

Acknowledgement

The path to reach the end of this thesis has been very long and troubled.

I would like to thank all the people that supported me and helped me over past months to complete my work.

Thanks to my supervisor Stefano Quer, for his experience, that provided relevant advice and interesting perspectives.

Thanks to Ciaran McCreesh and James Trimble, from Glasgow University: without their work, all my thesis would not have been possible.

Thanks to my sister Francesca, who actively helped me at every stage of my work, listening to my problems, proposing solutions and for being one of the few people that actually have read and helped to correct most of my thesis.

Thanks to Antonio, who often listened to me and helped me in understanding the more complex implementation aspects of the algorithms and alongside whom I frequently found myself fighting against the evil Nvidia APIs.

A special thanks to Costanza, who has never stopped supporting me over the past months, even in the days when things went wrong, successfully striving to revive my bad mood.

And finally, last but not least, also thanks to my mom, Laura, and all my other relatives who, despite not having any computer skills, have always shown an interest in my work and gave me their support to get to the end of my thesis.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Graphs are an extremely general and powerful data structure. They can be used to model, analyse and process a huge variety of natural phenomena and concepts by providing natural machine-readable representations. For instance, graphs can be used to represent chemical molecules, or road maps; in pattern recognition and computer vision, they are used to represent the patterns to be identified.

The aim of this thesis is to improve the implementation of the McSplit algorithm, considered the state of art to solve with constraint programming the Maximum Common Subgraph (MCS) problem, by using the powerful computational capabilities of modern graphic processors.

In the last twenty years, general purpose GPU computing has become more and more prominent in the computer science landscape, because of the progressive increase in power of the graphic units in consumer computers within the augmented efficiency and easiness to learn the APIs necessary to program such graphic units. This leads researchers to investigate new possibilities for implementing known algorithms in the GPU environment. Several different choices of developing platforms are possible when we approach to GPU computing.

Regarding the structure of this work, first we introduce some formal definitions and notation useful to better understand the details of our research. Chapter 2 is an overview on the Maximum Common Subgraph Problem (MCS) that is the main subject of this thesis. Chapter 3 presents and deeply analyses the McSplit algorithm, an efficient algorithm to solve MCS published in 2017. Then in Chapter 5 the Nvidia CUDA environment and its main APIs are introduced

and explained. This is a preparatory technical overview, useful to understand the modifications on the McSplit algorithm (described in Chapter 6) in order to make it executable on Nvidia GPUs. Then, in Chapter 7 we analyse experimental results, discussing the main issues which make this algorithm not suitable for a parallel implementation. Finally in Chapter 8 we conclude suggesting some possible different approaches and modifications to our work that would allow to achieve better results.

1.1 Formal definitions and notation

In this section we recall some basic definitions from the graph theory [Diestel 2018].

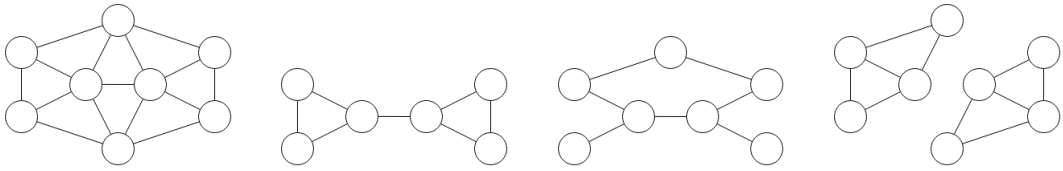
Traditionally, graphs are graphically represented by using dots or circles for vertices and lines for edges as in Figure 1.1.

Definition 1 *A **graph** G is an ordered pair $G = \{V_G, E_G\}$ where V_G is a set of entities called **vertex set**, and E_G is a set of two-elements subsets of V_G, called **edges set***

$$E \subseteq \{\{u, v\} : u, v \in V\}. \tag{1.1}$$

*Two vertices v and w are called **adjacent** if the tuples $\langle v, w\rangle$ or $\langle w, v\rangle$ belong to E_G.*

Definition 2 *A graph G is said **undirected** if, for any pair (v,w), vertex v is adjacent to vertex w if and only if vertex w is adjacent to vertex v. Otherwise the graph is **directed**.*



(a) Undirected graph with 8 vertices and 15 edges.

(b) Induced subgraph of (a).

(c) Non induced subgraph of (a).

(d) Non connected subgraph of (a).

Figure 1.1: Examples of graphs.

Except where otherwise stated, in this thesis, all graphs are considered *undirected.*

Definition 3 *The **order** of a graph G is the cardinality of V_G and the **size** of G the cardinality of E_G.*

Definition 4 *The set of all vertices adjacent to a vertex $v \in V_G$ is called **neighbourhood** of v, denoted $N(G, v)$. We specify with $\overline{N}(G, v)$ the **inverse neighbourhood** of v, being the set of all vertices not adjacent to v, excluding v itself. The cardinality of the neighbourhood of v denotes the **degree** of the vertex.*

Definition 5 *When the edges set E_G includes every possible unordered pair of vertices in V_G the graph is called **complete**. In such case the size of G is $K_n = n(n-1)/2$, where n is the order of G.*

Definition 6 *Labels, weights or other data might be associated with vertices or edges, in which case the graph is called **labelled**,*

Definition 7 *A graph H is a **subgraph** of graph G if it is composed by a set of vertices such that $V_H \subseteq V_G$. If H includes all the edges from G with both the endpoints in V_H, it is called an **induced** subgraph. Otherwise is called a **non-induced** or **partial** subgraph.*

Definition 8 *Two graphs G and H are **isomorphic** if it exists a bijective application $f : V_G \longrightarrow V_H$ between vertices from the two graphs, that preserves the relational structure. Which means that any two vertices u and v in G are adjacent if and only if $f(u)$ and $f(v)$ are adjacent in H.*

Definition 9 *Given a graph G, a **path** is a sequence of distinct vertices, where every consecutive pair is adjacent. We also allow the start and end vertices of a path to be the same vertex, in which case we have a **cycle**. The **distance** between two vertices is the length of a shortest path between them.*

Definition 10 *A graph G is **connected** if there exists a path between every pair of vertices. A connected graph with no cycles is called a **tree**.*

An often used synonymous for *vertex* in graph theory literature is *node*. However, in this thesis we will discuss about recursive algorithms that produce a search tree. With the aim of avoiding possible confusions, we will always use the term *vertex* to refer to elements in V_G, and reserve the term *node* to vertices of the solution search tree. In this thesis we will always consider only induced subgraphs, and we will also usually require them to be connected.

1.1.1 Subgraph problems complexity

Since we are talking about computational problems, in order to understand why we are interested to find efficient and fast algorithm to solve them, it can be useful to also briefly introduce some notation about computational complexity theory.

Definition 11 *In computability theory and computational complexity theory, a* ***decision problem*** *is a problem that can be posed as a yes-no question of the input values. A decision problem which can be solved by an algorithm is called* ***decidable****.*

The field of computational complexity categorises decidable decision problems by how difficult they are to solve. *Difficult*, in this sense, is intended in terms of the computational resources needed by the most efficient known algorithm that can solve a certain problem. The set of problems that are equally difficult to solve is called a *complexity class*.

Lot of complexity classes is defined, but, for what we concern, just four of them are here introduced: P, NP, NP-complete and NP-hard classes.

Definition 12 ***P*** *(Polinomial time) is a fundamental complexity class. It contains all decision problems that can be solved in polynomial by a deterministic Turing machine.*

This means that, given a problem p, and the size of its input n, the time needed to solve it, intended as the number of instructions that must be executed, is a polynomial function of n.

Definition 13 ***NP*** *(Non-deterministic Polynomial time) is a fundamental complexity class. It includes all decision problems solvable in polynomial time by a non-deterministic Turing machine.*

In computational complexity theory, some problems can be reduced into others by means of some transformation to input data, or some change to the problem statement. This can often be useful to solve problems, for example, an efficient algorithm is known to solve a certain problem p, and none is known to solve q, but some transformation to q is possible in order to reduce it to p. Then it is possible to solve p with the efficient algorithm and then transform back the result to adapt it to problem q. Some example of this concept will be presented in Section 2.2.

If a problem belongs to NP class and any other problem in NP can be somehow reduced to it, then it's called NP-complete.

Definition 14 *A problem $p \in NP$ is called **NP-complete** if every other problem in NP can be transformed (or reduced) into p in polynomial time.*

It is possible to *verify* a NP-complete problem but is not possible to *solve* it in polynomial time. For this reason, with problems of this class, we don't usually search an exact solution, because it would be too hard and time consuming to find it. Instead, we rather use some heuristic approach in order to search a reasonably good solution that approximate the optimal one following chosen criteria. Not always this approach is possible as we will better explain later.

Definition 15 *A problem p is **NP-hard** when, for every problem $q \in NP$, there is a polynomial-time reduction from L to H.*

Alternatively, we can say that NP-hard problems are all the problems that are *at-least* as difficult as the most difficult NP problems. This give also an alternative definition of NP-complete class, i.e. NP-complete problems are the set of problems that belongs both to NP and NP-hard classes.

Back to our context, problems that involve to find subgraphs with certain properties, often belongs to the NP-hard complexity class. This involves that computational complexity is such that the time required to solve problems with just one more vertex in input graphs can be exponentially higher.

The Maximum Common Subgraph problem, that we are going to introduce in Chapter 2, belongs to NP-hard complexity class, and, such as every other NP-hard problem, nowadays is still unknown an algorithm able to solve it in polynomial time. Moreover, the MCS problem belongs also to those problem for which

has been proofed the *hardness of approximation* [Kann 1992], that means that, unless is proven that P=NP, even the problem of its approximation still remain a NP-hard problem. Thus it is not also possible to find an MCS approximation in polynomial time, and research resources are only involved to search efficient algorithms to solve it exactly.

Algorithms we are going to illustrate are for this reason all supposed to calculate an exact solution, and they are very resources intensive in terms of memory and execution time.

Chapter 2

The maximum common subgraph problem

The Maximum Common Subgraph (MSC) Problem has been widely discussed in literature until now, in fact it was already well known in the seventies [Morpurgo 1971, Barrow and Burstall 1976, Bron and Kerbosch 1973, Cone, Venkataraghavan, and McLafferty 1977]. The problem has been defined in multiple ways, but in general all its formulations can be redirected to two main ones:

> *i) The **maximum common induced subgraph** of two graphs G and H is a graph that is an induced subgraph of both G and H, and that has as many vertices as possible;*
>
> *ii) Given two graphs G and H, the **maximum common edge subgraph** or maximum common partial subgraph, is a graph with as many edges as possible which is isomorphic to both a subgraph of G and a subgraph of H;*

In this thesis, we focus on the solutions to the type (i). Hence, for what we concern, MCS problem consists into finding for each graph the largest possible subgraph such that the two identified subgraphs are isomorphic. This problem belongs to NP-hard complexity class. The decision version of the problem can be expressed as follows:

> *i') Given two graphs G and H and an integer value k, does it exist a common subgraph to G and H with at least k vertices?*

In this formulation, that disregards whether k is the largest possible, the problem has been proved to belong to NP-complete complexity class [Garey and Johnson 2002].

2.1 Applications

This problem has a vast amount of practical applications in real life, and is therefore very carefully considered by the scientific community. In many circumstances, it can be necessary to compare graphs in order to check similarity or differences between the objects they model. Maximum common subgraph problems are one of the key steps in comparing graphs: to determine the difference between two graphs, we first have to find what they have in common [Kriege 2015].

These problems have been inspired scholars in different fields, such as chemoinformatics, and automatic circuit layouts. In the first one, the problem has application in pharmacophores mapping [Brint and Willett 1987]: algorithms have been developed in order to search maximal common substructure, for example common pharmacophores, in chemical molecules. Pharmacophores are the smallest structural element recognisable in a molecule, composed by functional groups dislocated in space that can interact with specific receptors and cause drugs' biological responses. Pharmacophores can be modelled by graphs, and, since they are present in several molecules, recognising them can be useful to investigate and predict which are the reaction of a human organism in contact with certain substances [Cao, Jiang, and Girke 2008].

Graph-based molecules can be compared also in chemical database management: subgraph isomorphism algorithms are used in substructure searching and recognition, where one wishes to identify all the molecules in a database that contain a user-defined pattern, or to find the correct place to put a new molecule based on similarities with near one. In this context, MCS algorithms provide an effective way of identifying the structural features common to different pairs of molecules [Willett 1999, Raymond and Willett 2002a].

Another interesting field of application of algorithms solving MCS problems is electronics. Always more often electronic circuits are assembled starting from off-the-shell spare components coming from the untrusted global market. The

lack of trust in these components requires additional validation of the components before use, to verify their functionalities and avoid hardware modifications and trojans that in some cases malicious producers add to their circuits [Li, Wasson, and Seshia 2012]. In fact, it is possible to model behavioural patterns of unknown circuits and known library components through graphs, and then applying techniques and algorithm to match isomorphisms between maximal subgraphs in order to reverse engineer the hardware layout.

2.2 Known approaches

The Maximum Common Subgraph problem has a long tradition in literature starting from the seventies. Various algorithms have been developed about graph and subgraphs isomorphism verification and common isomorphic subgraph research. In most of them we can identify two main approaches adopted in order to solve the problem.

Constraint programming

The first approach consists into applying traditional constraint programming based on backtracking, with branching and pruning strategies. In [Levi 1973], based on [Morpurgo 1971], such an algorithm is described, which makes use of the *maximal compatibility classes* to find the MCS; and in [Cone, Venkataraghavan, and McLafferty 1977] the use of this algorithm in comparing molecules has been discussed. [McGregor 1982] introduced a MCS algorithm that uses a branch and bound, and backtracking search. Each branch of the search tree corresponds to the matching of two vertices, and a bounding function evaluates the number of vertices that still may be matched so that the current branch is pruned as soon as this bound becomes lower than the size of the largest known common subgraph. [Krissinel and Henrick 2004] refined McGregor approach with a more efficient research. This approach has been taken over by the McSplit algorithm [McCreesh, Prosser, and Trimble 2017].

Maximal clique

The main alternative approach for solving the maximum common subgraph problem is to reduce the problem to finding a maximum clique (MC) in an association graph [Levi 1973, Barrow and Burstall 1976, Koch 2001, Raymond and Willett 2002b]. A *clique* is nothing but any complete subgraph in a given graph.

The association graph of two given graphs G and H is an undirected graph $G \triangledown H$ with vertex set

$$V_{G \triangledown H} = \{(v, v') \in V_G \times V_H : (v, v) \in E_G \Leftrightarrow (v', v') \in E_H\}.$$

Vertices in the association graph are named *matching nodes*, as each vertex $(v, v') \in V(G \triangledown H)$ denotes a match between the vertices v and v' from the input graphs. Thus, finding a maximum order clique in the association graph of G and H results into finding the maximum common subgraph between G and H.

As well as the MCS, the MC problem belongs to NP-hard complexity class; however, combined with a modern maximum clique solver [San Segundo et al. 2013], this is nowadays the best approach for searching the maximum common subgraph in certain instances of the problem, such as on labelled graphs [McCreesh, Ndiaye, et al. 2016].

[Shoukry and Aboutabl 1996] and [Schädler and Wysotzki 1999] also proposed new approaches to solve the MPC problem based on neural networks, a highly improving but very complex field of research in these years.

To conclude this overview on the known methods to tackle MCS problems, [McCreesh, Prosser, and Trimble 2017] proposed a new approach that is consistently over an order of magnitude faster than traditional constraint programming. It exploits some invariant properties of the CP search tree in order to reduce memory requirements and to allow stronger branching heuristics, which are cheaper in terms of time consumption.

In this thesis, we discuss some further improvements to this last approach, by implementing it in the CUDA environment.

Chapter 3

The algorithm

A new recursive, constraint programming (CP) algorithm to solve the Maximum Common Subgraph problem has been developed in 2017 by researchers from University of Glasgow. By exploiting some properties of the data structures used during the recursion, it improves backtracking and pruning operations in order to maintain the branching and filtering benefits of CP; at the same time, it obtains a consistent speed-up, compared to the state of art in constraint programming, up to an order of magnitude for the exploration of the solution space [McCreesh, Prosser, and Trimble 2017].

The new algorithm also allows to search for solutions over larger graphs, due to a lower memory usage. However, the memory consumption needs to be reduced even further in order to implement the algorithm in the CUDA environment (see Chapter 6).

McCreesh, Prosser, and Trimble 2017 extend their work and apply it also to directed and/or labelled graphs. However, for the sake of simplicity and code readability, in this thesis we will only consider *undirected* and *unlabelled* graphs (cf. Section 1.1 for definitions). Similar adjustments as the ones performed by the researchers from Glasgow can be applied to our work in order to provide compatibility with labelled and directed graphs.

The algorithm has been called McSplit, a pun from the author's name, C. McCreesh, the Maximum Common Subgraph problem (MCS), and one of the main aspects of the algorithm, that consists into using label-classes to mark vertices and to *split* them recursively to narrow the research until a solution is found.

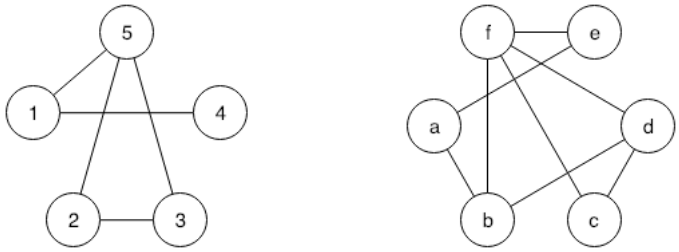


Figure 3.1: Example graphs G and H.

3.1 The McSplit algorithm

Let us consider two undirected and unlabelled graph G and H of order g and h respectively. We search for a mapping $M = \{(v_1, w_1), ..., (v_n, w_n)\}$ of $|M| = m$ vertex pairs, where $v_i \in V_G$ and $w_i \in V_H$ are distinct vertices from the input graphs, such that v_i and v_j are adjacent in G if and only if w_i and w_j are adjacent in H.

Given the largest mapping, namely $|M^*| = \max(m)$, the subgraph in G induced by $\{v_1, ..., v_n\}$ and the subgraph in H induced by $\{w_1, ..., w_n\}$ are isomorphic by construction and correspond to the maximum common subgraph of G and H.

The algorithm has a recursive structure based on a depth-first search. Starting from $\varnothing$ at each depth level, for each vertex $v_i \in G$, first it tries all the matches (v_i, w_i) with all the $w_j \in H$; then it also computes the potential subgraphs leaving the vertex v_i unmatched.

Let us analyse a simple example: consider the graphs G and H in Figure 3.1. They have a maximum common subgraph with four vertices; one of the possible solutions is the mapping $\{(1, a), (2, f), (3, d), (5, b)\}$. So the subgraph of G induced by vertices 1, 2, 3, 5 is isomorphic to the subgraph of H induced by a, b, d, f. Other possible mappings are illustrated in Figure 3.2.

One of the core aspects of the algorithm is the labelling of vertices during the progress of the search. Every time a new pair is added to the mapping, all the other vertices are assigned with a new label that keeps track of whether they are adjacent or not to every vertex already in the mapping. Back to the example, the algorithm first arbitrary chooses vertex 1 from G, and tries to match it with vertex a from H.

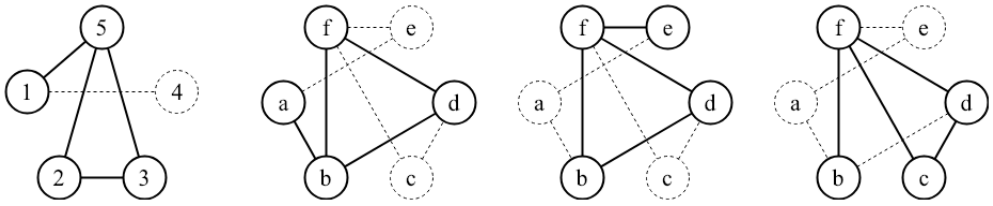


Figure 3.2: Example of different maximum common subgraph solutions.

(a) After mapping 1 to *a*

Mapping	Labelling of G		Labelling of H	
{(1,a)}	Vertex	Label	Vertex	Label
	2	0	b	1
	3	0	c	1
	4	1	d	0
	5	1	e	1
			f	0

(b) After mapping 2 to *d*

Mapping	Labelling of G		Labelling of H	
{(1,a),(2,d)}	Vertex	Label	Vertex	Label
	3	01	b	11
	4	10	c	11
	5	11	e	10
			f	01

(c) After mapping 3 to *f*

Mapping	Labelling of G		Labelling of H	
{(1,a),(2,d),(3,f)}	Vertex	Label	Vertex	Label
	4	100	b	111
	5	111	c	111
			e	101

Figure 3.3: The labelling applied in first three recursion of the algorithm.

Now each unmatched vertex in V_G is labelled according to whether it is adjacent to vertex 1, and each unmatched vertex in V_H is labelled according to whether it is adjacent to vertex a, as shown in Figure 3.3a. Adjacent vertices have label 1; non-adjacent vertices have label 0.

For each next recursion, the same approach is followed: at every recursion, the mapping M can be extended with a new pair (v, w) if and only if v and w have the same label. This property of mapping equal-labelled vertices from the two input graphs is the main core aspect of the algorithm.

The next step is to map another vertex of G with an unmatched vertex in H that shares the same label. Supposing that the algorithm chooses the vertices 2 and d, the mapping now becomes $M = \{(1, a), (2, d)\}$. Now, a two-character string is used to label unmatched vertices, such that the first character is the label of the first step and the second is the new label that states whether the vertex is adjacent or not to the newly mapped vertex. For example, vertex 3 is labelled with 01 because it is not adjacent to vertex 1 but it is adjacent to vertex 2, which are the vertices already in M (Figure 3.3b). Labels are given to unmapped vertices in $V(H)$ in a similar fashion, showing adjacency to matched vertices a and d.

In the following steps the invariant method is maintained: only vertices that share the same label can be mapped together and added to the mapping.

At each level it is possible to calculate an upper bound of the number of possible pairs that we can add to the mapping before backtracking and trying other mappings. For instance, in Figure 3.3c three different labels are used: 100, 101, and 111. The first two appear both only in one graph: since it is not possible to identify a matching element in the other graph, no new pairs formed with those vertices can be added to the mapping in the future. Label 111, instead, appears once in G and twice in H, so one new pair of vertices with this label can be generated before backtracking.

Thus, the upper bound of the mapping size is the sum of the smallest number of occurrences for each label that appears at least once in both graphs. A general

formula for this is

$$bound = |M| + \sum_{l \in L} min\Big(\big|\{v \in V_G : label(v) = l\}\big|, \big|\{w \in V_H : label(w) = l\}\big|\Big) \tag{3.1}$$

where L is the set of all labels used in both graphs.

3.1.1 The McSplit formalisation

The McSplit Algorithm has been formalised as shown in Algorithm 1 and implemented by colleagues from University of Glasgow, and published in McCreesh, Prosser, and Trimble 2017.

The main features of the McSplit Algorithm are *label-classes*, i.e. groups of vertices with the same label, and a recursive function (line 1), which has two parameters: $future$, that contains the list of available label-classes; M, that is the current mapping of vertices. A global structure, the *incumbent*, is updated during the recursions with the best solution found at each time.

At each run of the function `search`, first the global *incumbent* is updated if a greater mapping has been found so far; then the upper bound for the current search branch is computed according to (3.1), and if the bound has been hit (i.e. the bound is lower or equal to the size of actual *incumbent*, so any future search cannot improve it), the branch is pruned and the function returns (lines 3 and 4).

Now the proper research begins. First, a label class is selected from $future$ according to some decided heuristic[1], then a vertex v belonging to that label class is selected from G and excluded from following searches.

An iteration is performed on all vertices $w_i \in H$ that belong to the same label class. For each of them, we explore the consequences to add the pair (v, w_i) to the mapping. Another iteration is done on every label-classes in $future$: considering (v, w_i) as a new pair in the mapping, each label-class is now split in two new classes, according to whether the vertices belonging to it are adjacent to v and w_i.

The first class (lines 10 to 13) contains vertices in G adjacent to v and vertices in H adjacent to w_i. This is added to $future'$ if both sets contain at least one

[1]—tuning—-

Algorithm 1 The McSplit Algorithm

```
1:  function Search(future, M)
2:      if |M| > |incumbent| then incumbent ← M
3:      bound ← |M| + Σ_{⟨G,H⟩∈future} min(|G|, |H|)
4:      if bound < |incumbent| then return
5:      ⟨G, H⟩ ← SelectLabelClass(future)
6:      v ← SelectVertex(G, ⟨G, H⟩)
7:      for w ∈ H do
8:          future' ← ∅
9:          for ⟨G', H'⟩ ∈ future do
10:             G'' ← G' ∩ N(G, v) \ {v}
11:             H'' ← H' ∩ N(H, w) \ {w}
12:             if G'' ≠ ∅ and H'' ≠ ∅ then
13:                 future' ← future' ∪ {⟨G'', H''⟩}
14:             G'' ← G' ∩ N̄(G, v) \ {v}
15:             H'' ← H' ∩ N̄(H, w) \ {w}
16:             if G'' ≠ ∅ and H'' ≠ ∅ then
17:                 future' ← future' ∪ {⟨G'', H''⟩}
18:         Search(future', M ∪ {(v, w)})
19:     G ← G \ {v}
20:     future ← future \ {⟨G', H⟩}
21:     if G' ≠ ∅ then future ← future ∪ {⟨G', H⟩}
22:     Search(future, M)
23:     return
24: function McSplit(G, H)
25:     global incumbent ← ∅
26:     Search({⟨V(G), V(H)⟩}), ∅
27:     return incumbent
```

vertex. The same is repeated for non-adjacent vertices (lines 14 to 17).

A recursive call to `search` is performed with $future'$ and $M \cup \{(v, w_i)\}$, so a new search is performed with one more pair in the mapping and new label-classes that take into account this addition.

After each possible pairing between v and the vertices in H with the same label has been explored, a new recursion is performed considering solutions with the vertex v unmatched. v is removed from G and if it was the last considered vertex in its label-class, also the label-class is removed from $future$.

This formalisation could be hard to understand and to link to a practical example. In Section 4.1 the C language implementation of this procedure is analysed, making easier for the reader to follow the execution of the algorithm step by step.

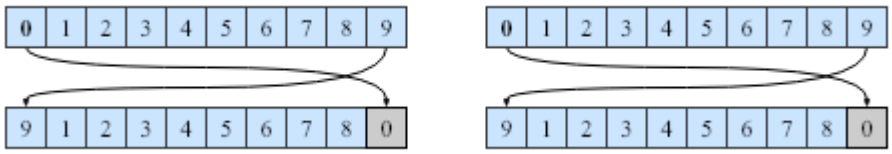
3.1.2 The bidomains system

The data structure for representing label classes is the most important aspect of the McSplit algorithm, the one that made us choose this algorithm to implement it in the CUDA environment. In fact, the labelling system is quite memory expensive: for each vertex we should store and keep track, at each recursion, of a string containing the label for that vertex. The larger the input graphs are, the less sustainable this solution would become. The system of *bidomains* developed by [McCreesh, Prosser, and Trimble 2017], based on the work of [Presa 2009] and [Bron and Kerbosch 1973] allows to significantly reduce the amount of memory necessary to represent label classes.

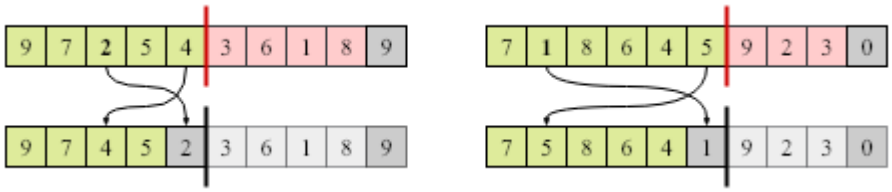
Given the complexity of a bidomain, let us analyse step by step the nature of the information that is necessary to store for the correct execution of the algorithm.

A particularity of label-classes is that, with the progress of the recursions, they remain self-contained, i.e. every time we split a label, we obtain two new labels with an equal prefix and just a 0 or 1 appended at the end. Thus, when we later need to backtrack, we just remove the suffix and put again together the vertices with same prefix.

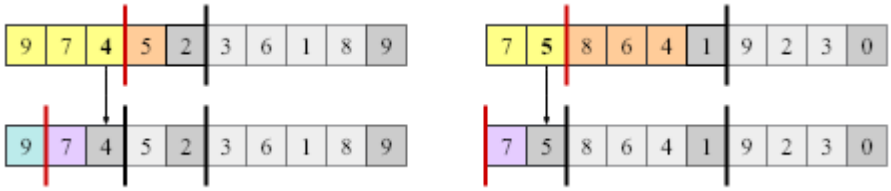
One of the common ways to represent graphs is through an array containing their vertices, and a matrix describing the adjacencies between them. When the vertices are labelled, and the set V_G is modelled with an array l, we can adjust the order of cells of l such that consecutive cells contain vertices belonging to the same label-class. By doing so, we can think label-classes as portions of that array.



(a) New pair selected $\langle 0, 0 \rangle$



(b) New pair selected $\langle 2, 1 \rangle$



(c) New pair selected $\langle 4, 5 \rangle$

Figure 3.4: With graph G on the left and H on the right, the figure represents an in-depth focus on label-splitting during a hypothetical execution of the algorithm.

In order to better explain how to work with label-classes in this way, let us consider a simple example with imaginary graphs G and H, both with ten vertices represented into two arrays, namely $left$ and $right$ (Figure 3.4).

Initially (Figure 3.4(a)), when no vertex has been selected yet, a single label-class exists in the $left$ and $right$ arrays (represented in *blue*). Suppose that the heuristic used to pick the vertices from the arrays to form new pairs is simply to select the vertex with the lowest value in the leftmost label-class. So, the first mapped pair $\langle 0, 0 \rangle$ is selected. In both arrays the vertex 0 is removed from future searches by swapping it with the last element of the label-class; also, the dimension of the label-class is decreased by one unit.

Now vertices in the *blue* label-class are rearranged according to their adjacency to vertex 0 (i.e., each adjacent vertex is moved before any non-adjacent one), and the label-class is split. Two new label-classes are therefore generated (*green* and *red* in Figure 3.4(b)).

Following the same heuristic for the next step and focusing on the *green* label-class, we select in each array the smallest vertex and we combine these in the second pair: $\langle 2, 1 \rangle$. As before, the two vertices are moved to the end of the respective label-classes, the vertices are rearranged and the new label-classes *yellow* and *orange* are created.

Now, the vertices 4 and 5 are mapped together and form the next pair $\langle 4, 5 \rangle$. They are moved to the end of the *yellow* label-class and this class is split into *light-blue* and *purple* (Figure 3.4(c)). Suppose that in the *right* array no vertices can be assigned to the *light-blue* label-class and only the purple one is generated. Since no matching is possible in the *light-blue* label-class, the algorithm is forced to stop and start backtracking, in order to analyse the consequences of choosing different pairs of vertices in the same label-classes and also to consider the other label-classes, such as *purple* or *orange* ones.

Label-classes described as above can be efficiently represented using the data structure *bidomain*.

```
typedef struct bidomain{
    int left_start, right_start;
    int left_length, right_length;
    bool is_adjacent;
} bidomain;
```

Source 3.1: Bidomain structure

A *bidomain* stores the first index of $left$ and *right* (the arrays containing the vertices of the two input graphs) where each single label-class begins and its length. Bidomains store also an additional information, that is whether the represented vertices are adjacent to the last mapped pair of vertices in the solution. Representing bidomains in this way, allows to discard the prefix part of labels and just store the final bit.

The bidomain b corresponding to the blue label-class in Figure 3.4 is $b = [0, 0, 10, 10, 0]$, where each component is respectively as follows:

- $b[0] = 0$ is the start index of the *blue* class on the *left* array.
- $b[1] = 0$ is the start index of the *blue* class on the *right* array.
- $b[2] = 10$ is the length of the *blue* class on the *left* array.
- $b[3] = 10$ is the length of the *blue* class on the *right* array.

- $b[4] = 0$ considered as a boolean value (0 = false, 1 = true), this value has the same role of bit "0" and "1" we appended to labels in the example represented in Figure 3.3.

Following the same logic, the *green* bidomain is $g = [0, 0, 5, 6, 1]$, the *red* bidomain is $r = [5, 6, 4, 3, 0]$, the *yellow* one is $y = [0, 0, 3, 2, 1]$ and so on.

Chapter 4

The McSplit Algorithm implementation

More than one implementation of the McSplit algorithm are provided by [McCreesh, Prosser, and Trimble 2017]. The first version, [Trimble 2017a], is a complete sequential version in C language, with several parameters and fine technical code used to properly handle different use cases for the algorithm; the second, [Trimble 2017b], is a simplified C language version of the first one, with less parameters, cleaner structure and static memory allocation, so that it's easier to start from here to explain the code. Actually, this version can also be more helpful than the formal pseudo-code (Alg. 1) to understand the main logic of the algorithm.

Finally, the last relevant version, [Trimble 2017c] (discussed in [McCreesh 2017] and [Hoffmann et al. 2018]), is a CPU parallel multi-thread version, written in C++ language. This version achieves very good performances, compared to sequential version, by means of a multi-threading structure that is quite complex, but it is not compatible with the execution in CUDA environment. A large part of our initial work was dedicated to understand this version and to look for an alternative multi-thread implementation with good performances, that would be suitable for a GPU porting.

Also a sequential version of the code is provided in C++ language in [Trimble 2017d], using almost the same logic of the C language version, but its syntax come from the C++14 standard, and it is often not trivial to understand. Given

our goal to write a plain C code we will omit to discuss this version.

For sake of readability, personal taste, and some small differences in notation for data structures and functions, we decided to write two new version of the code provided by authors, one sequential and one parallel, without altering their original logical structure or content, but using the same notation and C programming language for both the version. Our sequential version is almost identical to [Trimble 2017b], just little adjustments has been done to notation in order to improve readability and coherence with all other code versions. C language has been chosen because of Nvidia CUDA APIs are supposed to be used in C/C++ environment, and they make use of C-like syntax.

In the following highlights from each version will be provided.

Data structures

First of all we need to introduce the main data structures that will appear in the code. They will remain almost the same in both sequential and parallel implementation. Five C-language structures are presented in Source 4.1 and 4.2, we will refer to them using their re-defined short names.

Input graphs are modelled by `graph_t` in a quite common way (code 4.1): an `unsigned` value *n* is the degree of the graph; a two-dimensional `unsigned char` matrix describes adjacencies between vertices (if `v=3` is adjacent to `w=5` in the matrix `m` there will be a `"1"` in cell `m[3][5]`, a `"2"` is put on the diagonal for each vertex on which a loop is present); optionally each vertex can be associated with a label, stored exactly in the array `label`. For efficiency reasons, vertices of graphs are sorted based on their degree in decreasing order, such as lower values in the vertices array correspond to vertices with higher degree.

```
6  typedef struct graph_s {
7      unsigned int n;
8      unsigned char adjmat[MAX_N][MAX_N];
9      unsigned int label[MAX_N];
10 } graph_t;
```

Source 4.1: The `graph_t` structure: `MAX_N` is the maximum graph size supported, arbitrarily set, for the first versions, to 4096 vertices. Given the problem complexity, the value is fairly large enough.

From here on, we will often refer to graphs *G* and *H*, specially when treated as functions parameters, as `g0` and `g1`, or *left* graph and *right* graph. Hence, during

the execution of the algorithm, two more `unsigned integer` arrays are defined outside this structure to describe vertices of the input graphs. Such arrays are initialised with increasing integer values from 0 to $n-1$, so that each value in the array represents a vertex of the input graph. These two arrays are called `left` and `right`, respectively representing sets V_G and V_H.

What we want to retrieve, at the end of the algorithm, is a mapping between subsets of vertices of graph G and graph H. Thus, in code 4.2, first we define `pair_t` (line 1), which represent a mapping between a vertex $v \in V_G$ and a vertex $w \in V_H$. The two integer values `v` and `w` are values selected from `left` and `right` arrays. Now we can define a `mapping_t` (line 5) as a dynamic array of `pair_t` with *allocated* cells and *actually used* cells counters, respectively `size` and `len`.

```
1  typedef struct vtx_pair_s {
2      int v, w;
3  } pair_t;
4
5  typedef struct vtx_pair_list_s {
6    pair_t *vals;
7    unsigned len, size;
8  } mapping_t;
9
10 typedef struct bidomain_s {
11   int l, r, left_len, right_len;
12   bool is_adjacent;
13 } bidomain_t;
14
15 typedef struct bidomain_list_s {
16   bidomain_t *vals;
17   unsigned len, size;
18 } bidomain_list_t;
```

Source 4.2: Main data structures

Label classes, such they are described in Section 3.1.2, are also defined: inside `bidomain_t` structure (line 10) there is a bunch of integer parameters, their meaning is the same as described at the very end of Chapter 3: `l` is the initial index on `left` array where the label class begins, same for `r` on the `right` array; `left_len` and `right_len` are the respective lengths in terms of array's cells of label classes on the `left` and `right` arrays (i.e., any vertex into the left array from cell `left[l]` to cell `left[l+left_len-1]` belongs to the same label class, same reasoning for `right` array). Last parameter present in the structure is the boolean flag `is_adjacent`, with the same purpose of "0" and "1" used in label

classes such as they were presented in Figure 3.3.

Similarly to vertices mapping, we need to define lists of bidomains too. For this reason `bidomain_list_t` is also defined, in the same way of `mapping_t` (line 15). Given this brief highlight on data structures necessary to understand the code, we can now go deeper and analyse the core sections of sequential and parallel source code.

4.1 Sequential execution

The structure of the sequential code is quite simple. It is composed by just two main functions, one for the initialisation of necessary data structure, and one that is the proper recursive solver of the problem. In the following a code extract of the recursive function, namely `solve`, is presented, for a complete overview of the this sequential version of the code, refer to the original author's code, cf. [Trimble 2017a] or [Trimble 2017b].

`solve` function's parameters

The parameters of the recursive function are many, and most of them are passed by reference in order to keep them modified during the exploration of the recursion tree and exploit recursion properties. The two input graphs `g0` and `g1` are passed first, and to follow the vertices mappings storing the best solution found so far (the `incumbent`) and the currently considered solution (`current` solution); the parameter `domains` contains the list of all label classes selectable for the current recursion level, in form of bidomains, it is the corresponding variable of parameter *future* described in the pseudo-code of the McSplit Algorithm (Alg 1); finally arrays `left` and `right` are passed, they store a permutation of the indices of `g0` and `g1`, that depends on the recursion level and label classes, as illustrated in section 3.1.2.

`solve` function's body

The first operation performed by the function is to check if a better solution than the one currently stored into `incumbent` has been found. If the check is successful, such solution is stored into `incumbent`. Then, similarly to most of

recursive functions, `solve` proceeds with some checks intended to detect the moment in which the recursion has to stop. The upper bound of the current branch (i.e. the maximum number of pairs that is still possible to create given the current bidomains list) is computed by applying the (3.1) formula. If the bound, added to the current solution length, is not sufficient to overcome the incumbent length, it means that any additional calculation is useless, the branch is pruned and the function returns.

Then a heuristic method is applied to select a bidomain from the list (line 35), if none is found it means that the branch has been completely explored and, again, the function returns. Such heuristic consists into selecting the domain with the lowest max between the left and right length, i.e. the one in which the highest number between `left_len` and `right_len`, is the lowest. In case of tie, the class containing the higher degree vertices is selected [1].

```
26 void solve( graph_t *g0, graph_t *g1, mapping_t *incumbent,
27             mapping_t *current, bidomain_list_t *domains,
28             int*left, int*right){
29
30     if (incumbent->len < current->len)
31         set_incumbent(current, incumbent);
32     if (current->len + calc_bound(domains) <= incumbent->len)
33         return;
34
35     int bd_idx = select_bidomain(domains, left, current->len,
                        arguments.connected);
36     if(bd_idx == -1)
37         return;
38     bidomain_t *bd = &domains->vals[bd_idx];
39
40     int v = find_min_value(left, bd->l, bd->left_len);
41     remove_vtx_from_left_domain(left, &domains->vals[bd_idx],v);
42
43     int w = -1;
44     bd->right_len--;
45
46     for(int i = 0; i < bd->right_len +1; i++){
47         int idx = index_of_next_smallest(right, bd->r,
                        bd->right_len + 1, w);
48
49         w = right[bd->r + idx];
50         right[bd->r + idx] = right[bd->r + bd->right_len];
51         right[bd->r + bd->right_len] = w;
```

[1] Remember: vertices are sorted, so lower value in the array means higher degrees

```
52
53          bidomain_list_t *new_domains = filter_domains(domains,
                        left, right, g0, g1, v, w);
54
55          current->vals[current->len++] = (pair_t){.v=v, .w=w};
56          solve(g0,g1, incumbent, current, new_domains, left,
                        right);
57          current->len--;
58      }
59      bd->right_len++;
60      if (bd->left_len == 0) remove_bidomain(domains, bd_idx);
61      solve(g0, g1, incumbent, current, domains, left, right);
62  }
```

Source 4.3: function `solve` sequential implementation

As we already outlined in Section 3.1.1, the way we choose vertices from `g0` and `g1` is different. In line 40 and 41 a vertex from `left` array is picked and removed from his bidomain. The one with lowest value among the vertices in the selected bidomain is chosen. It is the vertex with the highest degree between the ones still present in the bidomain, and for this reason with the highest probability to allow more mappings in future recursions. The removal is simply performed by swapping the vertex with the one at the end of the bidomain, then reducing bidomain size, as illustrated in Figure 3.4.

Since we still not have picked any vertex from the right side of the selected bidomain, we initialise `w` with value -1, then we reduce the size of right domain by one (lines 43 and 44). We have to choose every possible vertices from the right bidomain, pair them with `v` and recur with increased solution, this operation simplifies the removal and insertion into right domain of `w` from time to time.

Now, from line 46 to 58, we iterate on vertices belonging to the right side of the selected bidomain (i.e. vertices of `g1` that can be matched with the selected `v`). The strategy is again to chose them following their degree order, selecting first vertices with lowest value and highest degree. Since they are not naturally ordered in the array, because their order may change between iteration due to recursions, every time we pass the last selected value of `w` to the function in order to pick the correct following one (line 47).

Once `w` is moved out of the right bidomain (lines 49-51), a new set of label classes is created, considering the just created pair $\langle$`v,w`$\rangle$. With this new set of bidomains, and after having added the new pair to the current solution (line 55), a

recursive call to `solve` is performed (line 56. `filter_domains` takes in input the *old* domains, the new pair, the input graphs (necessary to access their adjacency matrices) and the current state of `left` and `right` arrays. The function splits in a half every *old* label class depending on whenever vertices are adjacent or not to pair ⟨`v`,`w`⟩, and creates a new list of bidomains in which only ones with at least one vertex in both graphs are inserted. After recursion, pair ⟨`v`,`w`⟩ is removed from the solution.

When the consequences of pairing vertex `v` with every vertices in right side of bidomain has been explored with the recursions cycle, we have to consider the hypothesis in which vertex `v` is not present in the final solution. So right bidomain size is increased again (line 59, and another recursion is performed without modifying the bidomains, or the solution, but just removing vertex `v` from the left domain, such previously done in line 41.

Eventually, if the left part of the currently selected bidomain has been emptied, the latter is removed from the bidomains list so that next recursions will select a new one (line 60).

4.2 Parallel execution

The hardest issue encountered while attempting to efficiently parallelise the MCS problem with multi-thread solutions, is to correctly balance the workload between threads.

The basic scheme of the recursive function in the here presented multi-threaded version of the code is more or less the same of the sequential one. In source code listings of this section, some parts that are similar to sequential version have been omitted in order to allow an easier comprehension of the altered structure. As for previous version, for a more complete version of the code, refer to the original author's code, cf. [Trimble 2017c].

Before focusing on analysing code, it is appropriate to introduce some high level main aspects of this parallel implementation, then we will go step by step to lower technical details explaining some code highlights.

High level overview

The multi-thread solution of this version consists into delegating part of the `solve` function and its deeper recursion levels to a bunch of *helper* threads. They cooperate with the main thread to compute in parallel different iteration of the main `for` cycle of the function `solve` (described in Source 4.3, lines 46-58). Thus, a pool of threads is set up and kept running for the whole execution of the algorithm. Such pool owns a priority queue in which threads can put tasks they want to be helped to execute.

To better understand what a *task* is in our context, suppose that the `for` loop is moved out from the `solve` function, and put into two different functions.

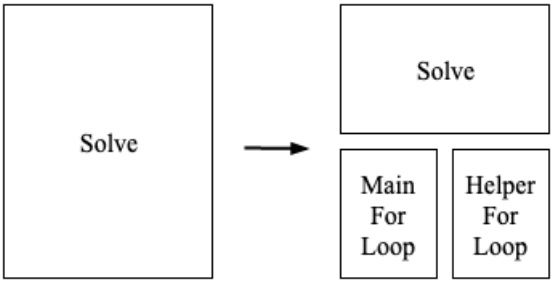


Figure 4.1: The function `solve` is splitted.

These two functions, namely `main_function` and `helper_function`, are designed such that they are independent and can be assigned to different threads to be solved in parallel.

Often, in the following, we will refer to threads executing `main_function` as *main* threads and threads executing `helper_function` as *helper* threads, but this names must not be confused with the actual first thread instantiated by the operative system and threads belonging to the thread-pool. We almost never use the term *main thread* with this acceptation in this chapter.

Indeed, in different phases of the algorithm, any thread can act as *main* thread and execute the `main_function` of a particular branch of the recursion tree. At the same time every other thread except for that one, will be considered *helper* threads.

In order to achieve independence between `main_function` and `helper_function`, `solve` function performs a copy of his relevant data structures before

return, and then passes the original data to the `main_function` that executes its `for` loop. The copies of the data structures are put together with a pointer to the `helper_function` inside a separate structure, forming a *task*.

The task is then put into a queue, and helper threads can pick it and cooperate with the main thread by executing the same `for` loop than the main thread, but in the `helper_function`, with the copied data structures.

In fact, though the two functions are independent from the point of view of data, they are actually linked by two atomic variables. The first one is used as a shared index in the `for` loop: main and helper threads can execute the same cycle, but distributing iterations without both executing the same ones. The second atomic variable is the size of the current incumbent, that is shared among each thread to increase the efficiency of the research.

Suppose, for example, that a thread *m* executes the `solve` function: after the preliminary operations, similar to the ones illustrated in Source 4.3 (lines 30-44), it creates an atomic variable *shared_i*, then it copies all the its relevant variables and creates a *task* containing such copies and a pointer to `helper_function`.

Thread *m* put the task into the thread-pool queue and starts executing the `main_function`. In a main thread, going from `solve` to `main_function`, is almost equivalent to continue the normal flow of the sequential version of the `solve` function (the one in Source 4.3). Suppose then, that thread-pool contains an idle thread *h*: it is now woke up and it picks the task from the queue and starts executing the `helper_function` with the copies of the data that main thread gave to it.

Remember that, for larger graphs, in the first few depth level of the recursion, each iteration of the `for` loop is very time consuming, because recursive calls to `solve` function are equivalent to solve the MCS problem on a graph with one less vertex than the original one. At the same time, the deeper you go in the recursion tree, the shorter become each branch, both because the available mappings are less, and it is easier for a branch to hit its upper bound and being pruned by the algorithm.

At such depth level it is not worth to create at each recursion copies of the data structures to apply the logic we described above, it would take more time to create them than to continue execute sequentially the branch until the end. For this reason, in order to not waste CPU-time copying data, and overload the

task queue in the thread-pool with too short tasks, causing massive slowdowns of the algorithm, a parameter `SPLIT_LEVEL` is set and tuned (a common value is in range 4-8). This parameter indicate the maximum depth in the recursion tree above which the solve function stops splitting itself and delegate iterations to other threads, but continues sequentially until the end of its current branch.

To achieve a higher level of parallelism, more than one helper thread can pick the same task and cooperate to solve it, also in different times. If at a certain time a thread completes its task, it returns to the queue in order to pick a new one, that may be already being executed by some other thread. Since they would use always the same shared atomic index, this fact would not be an issue.

The task objects are put in the queue following the order given by their position in the recursion three: a task generated at depth 1 will be put in the queue before a task generated at depth 3; in the same way, a task generate at depth 1 in the second iteration of the `for` loop will be put after a task generated at depth 1 in the first iteration.

In fact, besides the task, each node of the queue contains also an object *position*, that contains the depth `d` in which the task has been created, and, for each level from 0 to `d`, the iteration index of the for loop in previous recursions that led to that task: it is a kind of route in the recursion tree that identifies the position in the tree of the current task.

In Figure 4.2, a new task is being added to the queue. Its position is $\langle 2, [3, 1]\rangle$, it means that the task has been generated at depth 2 of the recursion, at level 1 the undertook iteration of the `for` loop was the third one, and at level 2, the first one. It is put in the correct position in the queue, after $\langle 2, [2, 2]\rangle$ and before $\langle 2, [3, 4]\rangle$, first considering depth and then values of previous `for` iteration indices in lexicographic order.

Above we said that the thread running the `solve` function create a copy of its relevant data structures before adding a task to the queue. This operation is done to guarantee data independence between threads. But, since more than one helper thread can pick the same task from the queue and execute it, they would not be independent by each other. Thus, a new *copy of the copy* of the data structures is performed at the beginning of the `helper_function`. In this way each thread can work on his own data, sharing with the other only the index `shared_i` and the incumbent size, and if no helper thread select the task from

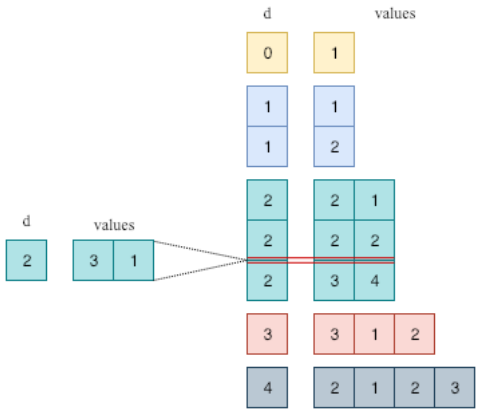


Figure 4.2: Detail of positions ordering when inserting a new node in the queue.

the queue, no other useless copies are created.

In the following we will better analyse how this task system has been implemented by [McCreesh, Prosser, and Trimble 2017]. Though the original code was in C++ language, with very different and advanced notation to handle threads and synchronisation, the logic of the task system has been consistently transposed in C language.

Data structures

The `threadpool_t` structure definition has been reported in Source 4.4. It includes data necessary to operate with threads: first the number of threads present int the pool (usually set to the number of physical or logical threads that can actually run in parallel on the processor); this number is also the size of the arrays of threads and the array of threads parameters (lines 3-4); by means of a mutex and a condition variable (lines 7-8)[2], the `threadpool_t` manage a queue of tasks that threads can pick and execute every time they become in idle state. An atomic boolean flag `finish` (line 10) is set to true when the algorithm terminate, and it is necessary to kill the threads belonging to the thread-pool and terminate the program.

```
1 typedef struct threadpool_s{
2     unsigned int n_threads;
```

[2]mutex and condition variables are from the POSIX standard. If necessary, a very good explanation of POSIX synchronisation objecs can be founs in [Kerrisk 2010, Chapter 30]

```
3     pthread_t *threads;
4     struct params_s ** params;
5
6     node_t *queue;
7     pthread_mutex_t general_mtx;
8     pthread_cond_t cv;
9
10    atomic_bool  finish;
11  } threadpool_t;
```

Source 4.4: Threadpool definition

Tasks, in this technical context, are defined in Source 4.5 (line 8) and consist in a pointer to the `helper_function`, and structure wrapping all necessary parameters for its execution, similar to parameters described for the `solve` function in Section 4.1. So once instantiated, threads repeatedly pick a task from the queue and execute the function pointed by `func` with its associated parameters pointed by `args`.

```
1   typedef void (*func_t)(args_t* d);
2
3   typedef struct position_s{
4     int vals[SPLIT_LEVEL +1];
5     int depth;
6   }position_t;
7
8   typedef struct task_s{
9     func_t func;
10    int pending;
11    args_t* args;
12  }task_t;
13
14  typedef struct node_s{
15    task_t *task;
16    position_t pos;
17    struct node_s *next;
18  }node_t;
```

Source 4.5: Threadpool definition

Since each task can be assigned to more than one thread, `task_t` also includes an integer counter `pending`, that keep track of how many threads are working on each single task. When a main thread add a task to the queue, it initialises the counter to 0. Each thread that start executing it, increment the counter and decrements it back when they finish. When the main thread exits from the execution of the `main_function` it wait until also any involved helper thread exits

from the respective helper function, then it remove the task from the queue.

The queue, defined in Source 4.5 (line 14), is structured as a linked list of `node_t` structures, each one containing a task, its position in the recursion tree, as we illustrated above, and the pointer to next node. The linked list structure is the preferred one for the queue, since we need to efficiently perform ordered insertion.

As we said, arguments passed to the `helper_function` (Source 4.6), are quite similar to ones of the sequential `solve` function. In addition, we also have an array of `mapping_t` storing separately the incumbents for each thread (line 17), this allows to reduce the need to synchronise them every time we update the incumbent using mutual accesses. Since it is a lot simpler to perform mutual access on a single, integer variable, an atomic unsigned value `global_incumbent` is defined: it is shared by every thread, and it is useful to keep threads synchronised at least on the size of the current best mapping found.

```
12 typedef struct args_s{
13   int depth;
14   graph_t *g0;
15   graph_t *g1;
16
17   mapping_t **per_thread_incumbents;
18   mapping_t *current;
19   bidomain_list_t *domains;
20   int *left, *right;
21
22   atomic_uint *global_incumbent;
23
24   position_t pos;
25   threadpool_t *pool;
26   int thread_idx;
27
28   atomic_uint *shared_i;
29   int i_end;
30   int bd_idx;
31   bidomain_t *bd;
32   int next_i;
33 }args_t;
```

Source 4.6: Threadpool definition

In line 24 we find the position of the current task, useful to determine the position to be passed to next recursions. In lines (25-26) data about thread pool and the index of the thread that is executing is stored, the latter just for debugging pur-

poses. Remaining arguments are variables useful for the continuation of the code flow from `solve` function to `main_function` or `helper_function`. In order to not weight down the notation, only one wrapper structure has been defined for both `main_function` and `helper_function`, thus not every arguments is useful for both of them. Anyway their purpose is trivial or it will be explained directly analysing the code in next subsubsections.

Core functions

As we said previously, recursion logic, bidomains system and graphs data structures are the same as the sequential version, thus, the aim of this paragraph is only to illustrate the functioning of the thread-pool and the related queue of tasks, that actually are the core aspects of this parallel implementation. Lot of the code is similar to Source 4.3, thus some parts of the source code extracts, that we will present here, will be reduced or replaced by comments. First of all a short highlight of the `helper_function` is presented in Source 4.7.

```
1  void helper_function(args_t *args){
2      int next_i = atomic_fetch_add(args->shared_i, 1);
3    //create a new copy of the copy of relevant data structures.
4    for (int i = 0 ; i < args->i_end; i++) {
5        if(i != args->i_end - 1){
6            // select a vertex w and make a new pair
7            // ...
8              if (i == next_i) {
9                  next_i = atomic_fetch_add(args->shared_i, 1);
10                 if (args->depth > SPLIT_LEVEL) {
11                     solve_nopar(/*...*/); // continue
                         sequentially until the end of the branch
12                 } else {
13                     position_t new_pos = args->pos;
14                     new_pos.depth = args->depth;
15                     new_pos.vals[new_pos.depth] = i + 1;
16                     solve(args->depth + 1, new_pos, args->pool);
17                 }
18             }
19         } else {
20             // vertex v is left unmatched
21             // ...
22             if (i == next_i) {
23                 if (args->depth > SPLIT_LEVEL) {
24                     solve_nopar(/*...*/); // continue
                         sequentially until the end of the branch
25                 } else {
```

```
26                      position_t new_pos = args->pos;
27                      new_pos.depth = args->depth;
28                      new_pos.vals[new_pos.depth] = i + 1;
29                      solve(args->depth + 1, new_pos, args->pool);
30                  }
31              }
32          }
33      }
34  }
```

Source 4.7: The cyclic function executed by every thread in the thread pool.

An important difference between Source 4.7 and the sequential version of `solve` that is worth noting, is the structure of the `for` loop: here we extended the loop by one iteration, such that it also includes the recursion in which vertex `v` is left unmatched in the solution. This modification is helpful for the distribution of recursion between threads.

Proceeding with order, in line 2 the shared index is immediately read and incremented, so that the running thread reserves for itself at least one iteration of the loop. Then the *copy of the copy* of data structure passed by argument is performed, and the loop begins. New pairs are created for each iteration, with vertex `w` being swapped each time with the one at the end of the right bidomain, such that permutations of vertices in `right` array is maintained the same as in the sequential version. Though here recursions, that explore consequences of adding each pair to the mapping, are performed only in the iteration identified by the shared index. Same rule is applied for the last iteration in which vertex `v` is left unmatched.

Recursions, for levels deeper than `SPLIT_LEVEL` are not performed through `solve` function, but through `solve_nopar`, that nothing is but the standard sequential version of the `solve` function, and thus not illustrated here.

The `main_function` is not reported here, because it is almost identical in the structure as the `helper_function`. The main difference is that the latter creates new copies of the data structures before executing the `for` loop, while the `main_function` does not.

```
34  void solve (int depth, position_t position, threadpool_t *pool,
                        ...){
35    /**
36     *  Same recursion stop controls and bidomain operations
37     *  as the sequential solve function
38     */
```

```
39    atomic_uint *shared_i = malloc(sizeof *shared_i);
40    atomic_store(shared_i, 0);
41    int next_i = atomic_fetch_add(shared_i, 1);
42    int i_end = bd->right_len + 1;
43    bd->right_len--;
44    args_t *main_args = wrap_args(depth, position, pool,
                        shared_i, i_end, ...);
45    if (depth <= SPLIT_LEVEL){
46      args_t *helper_args = wrap_args(depth, position, pool,
                        shared_i, i_end, ...);
47      run_in_threadpool(position, pool, main_function,
                        helper_function, main_args, helper_args);
48    } else {
49      main_function(main_args);
50    }
51  }
```

Source 4.8: Modified `solve` function for the parallel implementation.

As illustrated in Source 4.8, the new, parallel, `solve` function starts with more or less the same code of the sequential version that is not entirely listed. Then, before the point in which, in the sequential version, should begin the `for` loop, the function initialises the shared atomic index `shared_i` and wrap the arguments for the `main_function`. No copies are performed in this passage. Now, if the recursion depth is higher than the `SPLIT_LEVEL` parameter, the function will simply execute the `main_function` and continue sequentially until the end of the branch (in these circumstances also `main_function` would execute `solve_nopar` instead of `solve`). Otherwise, if depth is lower than `SPLIT_LEVEL`, then also arguments for the `helper_function` are wrapped into an `args_t` structure. Here a copy of data structures is stored and function `run_in_threadpool` is called.

```
52  void run_in_threadpool(position_t pos, threadpool_t *pool,
                        func_t main_function, func_t
                        helper_function , args_t *main_args,
                        args_t *helper_args){
53    /* create a task for the helper function */
54    task_t *task = create_task(helper_args, helper_function);
55
56    pthread_mutex_lock(&pool->general_mtx);
57    enqueue(&pool->queue, pos, task);
58    pthread_cond_broadcast(&pool->cv);
59    pthread_mutex_unlock(&pool->general_mtx);
60
61    (*main_function)(main_args);
```

```
62
63    pthread_mutex_lock(&pool->general_mtx);
64    while(task->pending != 0) pthread_cond_wait(&pool->cv,
                        &pool->general_mtx);
65    dequeue(&pool->queue, task);
66    pthread_mutex_unlock(&pool->general_mtx);
67  }
```

Source 4.9: Function called to add tasks to the thread pool queue.

The function `run_in_threadpool`, in Source 4.9, first creates a new `task_t` containing helper agruments and the pointer to `helper_function`. Then it acquires a mutual exclusion lock on the queue and add to it the new task. Immediately a signal is sent to every thread waiting on the condition variable of the thread pool and the lock on the queue is released.

From this moment, every thread eventually in idle, will be woke up and can start executing the `helper_function` included into the new task. Also the thread executing `run_in_threadpool` starts executing `main_function`.

When `main_function` returns, it means that no other iteration of the `for` loop has to be performed for the relative `solve` function. The thread acquires again the lock on the queue, and remove the task from the queue, after waiting for any helper thread still needing to complete its iteration.

A fundamental role in this system is of course played by the cyclic function executed by each thread belonging to the thread pool. The integral code of the function is reported in Source 4.10.

```
1  void* thread_func(void* params){
2    threadpool_t *pool = ((param_t*)params)->pool;
3    while(!atomic_load(&pool->finish)){
4      pthread_mutex_lock(&pool->general_mtx);
5      bool did_something = false;
6      for(node_t *node = pool->queue; node!= NULL;
                        node=node->next){
7        task_t *task = node->task;
8        if(task!= NULL && task->func!=NULL){
9          func_t f = task->func;
10         task->pending++;
11         pthread_mutex_unlock(&pool->general_mtx);
12
13         (*f)(task->args);
14
15         pthread_mutex_lock(&pool->general_mtx);
16         task->func = NULL;
17         if(--task->pending == 0)
```

```
18              pthread_cond_broadcast(&pool->cv);
19            did_something = true;
20            break;
21          }
22        }
23        if ((! did_something) && (!atomic_load(&help_me->finish)))
24          pthread_cond_wait(&help_me->cv, &help_me->general_mtx);
25        pthread_mutex_unlock(&help_me->general_mtx);
26      }
27    }
```

Source 4.10: The cyclic function executed by every thread in the thread pool.

The function is structured as an endless `while` loop that is stopped only at the end of the algorithm, after the first `solve` function has returned in the `main` function of the program. At each repetition of the cycle, various execution flow can be executed by threads. Let us analyse them separately.

i) The task queue is empty: suppose to be at the beginning of the program, when the thread pool is first initialised, but the algorithm still is not started, each thread will start the cycle when the task queue is still empty. In this situation each of them acquires the lock on the queue (not everyone at the same time, of course), sets to false boolean flag `did_something` that states if the thread has actually executed some task in this cycle or not, and then tries to pick a task from the queue. Since the latter is empty, the whole central `for` loop is skipped without doing anything, and the thread blocks on the condition variable waiting for someone to put a task in the queue and wake it up [3].

ii.a) The task queue contains tasks that no thread has still executed: when `run_in_threadool` function is executed for the first time, the the main thread add a task to the queue and calls `pthread_cond_broadcast`. Each thread of the thread pool that in that moment was waiting the condition variable (Source 4.10, line 24) is released. The first thread that manage to acquire the lock proceeds now repeating the `while` cycle. It set to false the flag `did_something`, and pick the task from the queue. Supposing that it is a fresh new task, both its reference and the reference to `helper_function` included in it should be not `NULL`. The check at line 8 it crucial but will be clear after the explanation of the remaining of the function.

[3] POSIX mutex and Condition variable objects are supposed to be well known. If necessary, for a very good explanation of such arguments, see [Kerrisk 2010, Chapter 30]

The pending counter is increased (line 10) to inform the main thread that a helper thread is going to execute the helper_function and the latter is executed (line 13). In order to understand the reason for which the helper_function pointer inside the task is set to NULL after the execution of the function, it is important to recall that the the function basically consists into a for cycle that is shared among threads. Thus, if the function returns for one thread, it means that no more iteration have to be executed at all, and soon or later also every other thread executing the same function will return, the exact time depends only on how long they will it take to them to finish their last iteration. For this reason the actually executing task, should no longer be executed by any other thread. But the only entity allowed to remove it from the queue is the one that put it in the queue during the execution of run_in_threadool and this will happen only after the last thread finishes executing it, i.e. when the pending counter of the task reaches 0. In fact, the next instruction in thread_func is to decrease the pending counter and, if value 0 has been reached, to signal the condition variable so that main thread can remove it from the queue. The did_something is now set to true, the for loop is aborted and before waiting for other task to be inserted in the queue, the while cycle is restarted trying to pick a new task from the queue.

ii.b) The task queue also contains already executed tasks: the tasks queue is ordered by position_t, and thus tasks are executed and completed following the order of the recursion. Any completed task that is still being executed by some thread, will be at the beginning of the queue. With a NULL pointer instead of the helper_function pointer, such tasks will be simply skipped by any thread scanning the queue searching for a task to execute. When a task with func pointer not NULL is found, the execution continues as the point *ii.a)* illustrates. If no task is found, the execution continues as in point *i)*.

The thread pool structured in the way we just illustrated allows to obtain a very good balance between the execution time of each thread: any time a thread finishes a branch, it can go through the tasks queue and start helping other threads with their branches. At the end each thread will have had very short idle time.

Chapter 5

The CUDA API

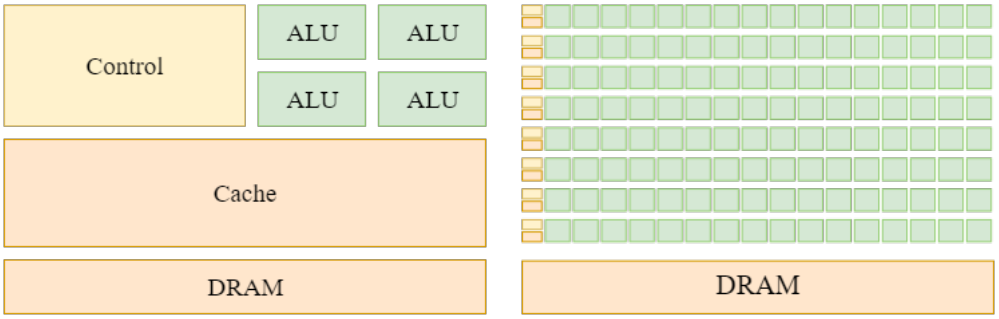
The Compute Unified Device Architecture, usually shortened with the acronym CUDA, is a parallel computing platform, developed by Nvidia Corporation since early 2000s and first released in 2007. It provides application programming interfaces (APIs) for general purpose programming on CUDA-enabled GPUs, i.e. graphic cards produced by Nvidia and belonging to specific product lines.

CUDA allows programmers to exploit the different parallel architecture of graphic processing units for certain categories of programs that require massive computing power.

Graphing computing usually involves large matrices modelling pixels on the screen, on which lots of heavy mathematical calculations have to be quickly computed in order to create new images that have to be shown on the screen with fast refresh rates. This mechanism offer offers high level of data-parallelism, because such mathematical operations can be done independently in parallel on each pixel.

Driven by the pressing market demand for real-time, high-definition 3D graphics, GPUs development has been aimed in past decades to increment performances in this direction. In fact modern retail GPUs can reach tens or even hundreds of teraFLOPS of computing power, that compared to CPUs, that are usually still measured in gigaFLOPS, represent quite huge numbers. Compared to CPU ones, more space on GPU chips is devoted to computing units than control units and caches, as illustrated in Figure 5.1.

Specific types of programs can perform particularly well on GPUs. The graphic



(a) CPU (b) GPU

Figure 5.1: CPU architecture overview, compared with GPU

processors architecture is aimed to maximise efficiency in high mathematical data-parallel programs, i.e. programs that execute lots of mathematical instructions on large data sets.

With an accurate design aimed to maximise parallelism of instructions, the performances obtained by running the code on a GPU instead of CPU are very impressive. In the following there will be a brief description about main concepts of the hardware architecture of Nvidia GPUs, among with some guidelines to produce code that can accomplish the efficiency and high parallelism standards necessary to obtain good performances in GPU.

Writing software that is able to run well on a graphic cards is a matter of lower level programming compared to modern programming on CPUs, that always more often makes use of high level languages that hide most of the hardware interaction to the programmer. Indeed, in traditional programming, programmers can write code without care much about hardware structure of processors, both because processors are a very heterogeneous family of devices with different architectures and it would be impossible to take into account of each difference in the code, and compilers are increasingly efficient at instruction translation and rearranging making high quality executables able to perform well on lots of devices. GPU programming is similar to programming for embedded devices: you cannot rely only on compilers, you have to know very well the architecture on which the code will run, arguments such registers management, code compactness or memory saving cannot be ignored.

Programming tools and environments are still not well consolidated to help

programmers with CUDA or other parallel computing platforms, thus technical knowledge is needed before approaching to GPU programming. Most of architectural information included in this chapter come [Cook 2012] and [Sanders and Kandrot 2010]. Any technical detail about APIs, such versions and specifications, can be found and verified at [Nvidia 2018b].

5.1 Hardware description

In past years CUDA micro-architectures evolved through several main versions, usually named as famous historical personalities, important for the field of computing technologies, such as Alan Turing, Nikola Tesla, Blaise Pascal and others. Features implemented by various versions are called *compute capabilities* or *compute levels* and evolved through different versions from 1.0 of first Tesla micro-architecture to 7.5 of the new Turing micro-architecture, being released at the time of this writing. The most of the computing features encoded with such version numbers depend on specific hardware units. Thus, compute levels are fixed for each device: usually, to upgrade the CUDA compute capability of your GPU, you cannot do anything but buy a more recent card. First versions 1.x still was on an uneven development phase, and did not include basic operations such as atomic operations on 32 bit integers in global or shared memory, or support for double precision floating-point operations (see Table 5.1), so we will not take them into account.

From compute capability 2.0 and higher (i.e. the Fermi micro-architecture), some important features have been consolidated. Successive versions just introduced some advanced features about memory handling, textures or other graphic tools, and increased efficiency and computing power. A deeper resumee about features included in each compute level is presented in Section 5.1.2. In any case, the hardware basic scheme has remained more or less the same over the year. Thus, in order to make a simpler description we will initially focus only on Fermi micro-architecture, released in 2010.

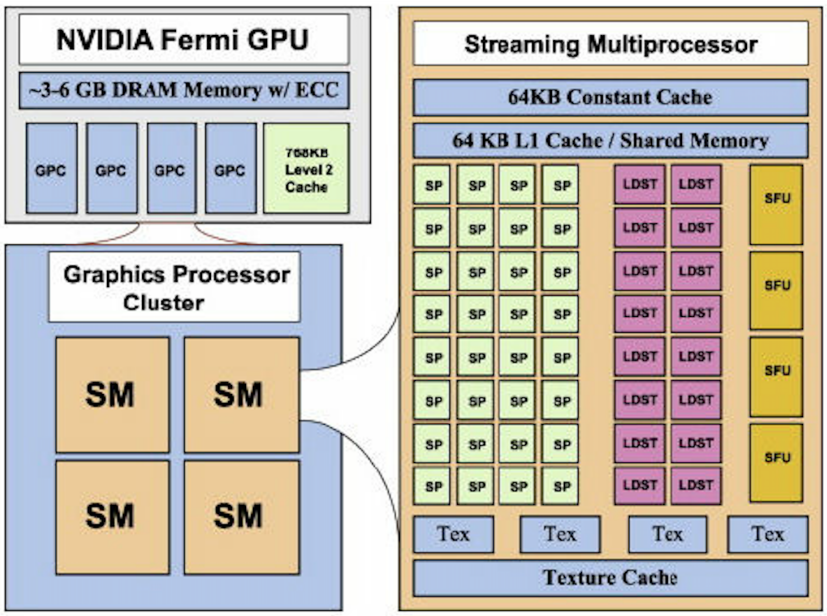


Figure 5.2: Fermi hardware micro-architecture scheme.

5.1.1 Nvidia GPU micro-architecture

As Figure 5.1 shows, the GPU hardware is radically different from CPU one. More closely, in Figure 5.2 is illustrated the general structure of one of the first Fermi micro-architecture, released by Nvidia in 2010. This high level structure holds also for graphics units produced by other manufacturers such AMD. It's possible to notice that a GPU basic scheme is something like an array of units called *streaming processors* (SP), plus some other computing device on the borders. In Nvidia devices, they are grouped in larger groups called *streaming multi-processor* (SM).

Each SM is composed by several Streaming Processor (also referred by Nvidia as *CUDA core*), 8 for the first released Tesla G80 GPU, 32 or more in later versions. According to the micro-architecture design, in order to scale performances is enough just adding more SMs, or more cores per single SM. The scaling potentiality of the hardware is important because in past decades, when consumer CPUs left the single-core architecture and started being composed by two or four cores, programmers had to rewrite lot of software in order to exploit new multi-thread features of processors. CUDA and other parallel platforms born already with the concept of having lots of processing units available, and code is not

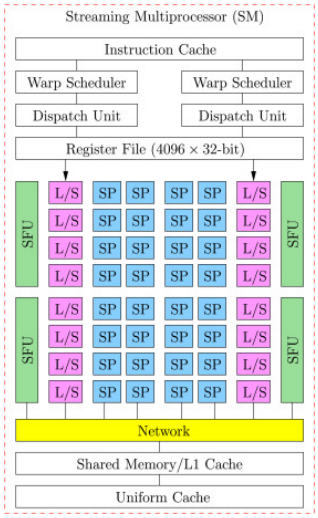


Figure 5.3: Fermi Streaming Multiprocessor.

meant to be written depending on the actual number of cores available. Same code should be suitable both for entry level cards, with few hundreds of cores, and high levels GPUs, featured with up to some thousands of cores. This approach allows to take advantages of more next generations hardware without underuse it and without the need of correcting code each time.

In Figure 5.3, a more precise representation of a Fermi streaming multi-processor shows the presence of other modules such as two *warp scheduler* that, together with *dispatch units*, are in charge to issue to cores the *warps*, that actually are groups of threads and will be better discussed later. Then a *load/store unit* that computes input and output memory addresses on behalf of each core, and *SFUs*, specific chips that are decoupled by cores and are in charge to compute complex operations such as sine, cosine and root square while cores execute other tasks. One or more level of cache memory are present for each multi-processor.

5.1.2 CUDA Compute capabilities

Compute capability, also referred as *compute level* or *SM version* of a GPU is expressed by a version number that represent the hardware features that such device can support. It is composed by a *major revision number X* and a *minor revision number Y*. Devices belonging to the same micro-architecture series, share the same major number, while the minor numbers represent incremental im-

Technical specifications	Compute capability															
	1.0	1.1	1.2	1.3	2.x	3.0	3.2	3.5	3.7	5.0	5.2	5.3	6.0	6.1	6.2	7.x
Integer atomic functions on 32-bit in global mem.	No	Yes														
Integer atomic functions on 32-bit in shared mem	No				Yes											
Integer atomic functions on 64-bit in global mem.	No				Yes											
Double precision floating-point operations	No				Yes											
Integer atomic functions on 64-bit in shared mem.	No				Yes											
Atomic addition on 32-bit floating-point in global and shared mem.	No				Yes											
3D grid of thread blocks	No				Yes											
Dynamic parallelism	No							Yes								
Atomic addition on 64-bit floating-point in global and shared mem.	No												Yes			
Tensor core	No															Yes

Table 5.1: A list of the main CUDA features that came out with the progress of compute capability versions.

provements and feature additions in the same micro-architecture.

Compute capability 1.x is from the first Tesla series (2006-10), 2.x for Fermi (2010-12), 3.x for Kepler (2012-14). 4.x devices never came out on the market, so from 3.x we go up to 5.x compute capability for Maxwell micro-architecture (2014-16), 6.x for Pascal (2016-18) and, finally, major revision number 7 came out first on the professional series of GPUs for workstations, using Volta micro-architecture (2017) and then with GPUs for the general market Turing in 2018. At the time of this writing, the actual compute capability available on market is 7.5, for GPUs belonging to Nvidia GeForce RTX 20X0 series, Titan RTX and GTX 1660/1660Ti.

A huge number of techincal details varies depending on compute capability of each device. Thus describing each of them would only result in a very large meaningless table. In Table 5.1 some of the more relevant and high level differences between each compute capability are reported, and other will be presented in the remaining of the chapter when analysing each different aspect of CUDA programming. For more precise information and a complete comparison between compute capabilities, see [Nvidia 2018a].

Reading the table, it is easy to understand the reason we chose to start discussion from compute capability 2.0 of the Fermi micro-architecture. In fact, several

important operations, such 32-bit atomic operations or even simple floating point operations, still were not supported.

Only in much more recent micro-architecture Pascal, with compute capabilities 6.x got enough harder resources to perform also up to 64-bit floating point operations in one single atomic operation in both global and shared memory. In the following section, such memory types will be discussed.

5.2 Memory

Inside each streaming multi-processor there are several different memory spaces: we have global memory and local memory, shared memory, different levels of cache, texture and constant memory, and finally, a large register file.

Each of those types of memory has different sizes, access policies and speed. In the following a short presentation of main ones will be provided.

Global and local memory

Even if they are configured in the same computing system, CPUs and GPUs have separate memory units and address spaces. Global memory resides into the main memory of the GPU device and correspond to the CPUs main memory. It can be accessed by any thread running on the GPU and also, with specific functions, by the CPU. It is used as the main communication tool between CPU and GPU. Usually, the CPU copies some data into the GPU's global memory, then it starts a task so that GPU can works on such data. Global memory is the slowest between every different types of memory on the GPU, but also the greatest: its size can vary from one model to another, but for modern cards it is usually expressed in terms of few gigabytes.

Local memory resides on the same device memory as the global one. An amount of local memory is reserved on per-thread basis, so it can be accessed only by the thread to which it has been assigned and can be used to store large intermediate data during computing. The speed of local memory is the same global memory one, but the size is significantly smaller: 16KBytes for compute capability 1.x and 2.x, and 512KBytes for higher levels.

Constant and texture memory

Constant memory is a read only memory space, available from each running thread on the GPU. It resides on the device memory, but has a separate cache space directly on the chip, so also accesses to constant memory can be much faster than global and local memories. It can be written only from the CPU using specific functions provided by the CUDA API. The size of constant memory is set to 64KBytes for each compute capability.

Similarly to constant memory, texture memory is a portion of the main device memory, with a different address space and optimised for storing and fetching texture objects. It has a dedicated on-chip cache, so accesses to texture memory are usually faster than local and global memory. Specific functions have been designed by Nvidia to allow efficiently fetching textures from this memory space. Since it has been optimised for graphics purposed, we will not use the texture memory in our work.

Constant memory, texture memory and global memory are the only types of memory persisting between different kernel calls, i.e. the functions that CPUs can launch in the GPU environment (see Section 5.4).

Shared memory

A chunk of fast memory, namely *shared memory*, accessible only inside a Streaming Multiprocessor and accessible by all threads belonging to the same block [1], can be used to share data between threads, as a programmer-managed cache. Shared memory reside directly on the chip, this is much faster than global and local memory, but its size is strongly limited by engineering aspects. In fact, shared memory available for each block of thread, is limited to 48KBytes for every compute capability but 7.x, in which it can be configured by programmers up to only 96KBytes.

Separate busses are present into each SM in order to independently access to the other kinds of memory present on the chip: constant, texture and global memory. Actually, the first two are just special views on a separate chunks of the global memory, with special rights and access policies. The global memory has more or less the same purpose of the CPU main memory, it usually is composed

[1] blocks of threads will be presented in Section **??**

by some blocks of GDDR memory that is a faster version than the memory used for CPUs and can give a bandwidth up to 10 times faster than normal DDR[2].

Registers file

The register file is a relatively large memory space that resides directly on the chip, the closest possible to each processor. It stores registers that are used in each thread. Differently to CPU threads, from compute capability 3.2, in the GPU threads can have up to 255 registers. But the main difference between CPU and GPU registers is not their number: the register file in GPU contains separate copies of registers for each thread. Thus, context switches[3] are simply performed by selecting which register must be mapped with the computing units, instead of having to swap in and out from the main memory each time, as it happens in CPUs, wasting several hundreds of clock cycles.

Moreover, GPUs use many threads to hide stall situations during memory fetches and other blocking operations. Every time a thread needs to block waiting that a variable is fetched from memory, or an arithmetic calculation is performed, the scheduler perform a context switch and the processor continues working executing another thread. Thus fast context switches are crucial for GPUs.

5.3 Serial and parallel code

Generally speaking, every modern processor has more than one processing unit inside (i.e. cores), so they can perform more operations at once. But in order to actually exploit this feature it is necessary to write code that can be executed in parallel. This is not always possible. In fact, algorithms can vary significantly in how, and how much, they can be parallelised. Some of them are completely unable to execute in parallel. Such problems are composed by steps that require as input data that depends on the output of a previous step, forcing the program

[2]GDDR and DDR stands for Graphic Double Data Rate and Double Data Rate respectively. These memory technologies allows read/write operation on the memory on both the system clock sides, up and down, leading to much faster memory accesses.

[3]context switches are the process of storing the state of threads, so that they can be restored later. This allows multiple thread to share a single processing unit, and is an essential feature of any parallel device

to execute in a serial flow. Indeed, one of the most difficult aspect of parallel programming is to be able, when possible, to think and write a problem in terms of instructions that are as independent as possible one with each other.

Algorithm 2 Non parallelisable alg.

```
1: function Foo_1()
2:     a ← {1, 2, 3, 4, ...}
3:     for i ∈ [1 − 9] do
4:         a ← a[i − 1] + i
       return a;
```

Algorithm 3 Parallelisable alg.

```
1: function Foo_2()
2:     a ← {1, 2, 3, 4, ...}
3:     for i ∈ [1 − 9] do
4:         a ← a[i] + 2
       return a;
```

As a very easy example of this concept, the Algorithm 2 above is a non parallelizable algorithm, in fact each iteration of the *for* loop depends on the previous one in order to compute the new value. On the other hand, Algorithm 3 is a parallelizable algorithm because each step of the loop is independent from any other step, so it is possible to execute all the iterations at once concurrently, of course, as long as you have enough cores on which run the code. Anyway these concept should be clear enough to programmers who already had some experience in *traditional* multi-thread programming

5.3.1 From multi-core to many-cores

When we talk about many-core devices, such as GPU devices, we're talking about processors with a number of cores around hundreds or thousands, much more than 4-8 cores inside modern average CPUs. The basic concept of parallel programming is the idea of *thread*, that is a single flow of serial execution of instructions through the program that runs on a processor core. Several threads that run in simultaneously on more processing cores make up a parallel program. This should be a familiar concept to any programmer that already had experiences with traditional multi-core parallel programming on CPUs.

As we said CPUs are composed by a small number of powerful cores and, unlike GPUs, follow the MIMD (Multiple Instruction - Multiple Data) scheduling model, in which they can run multiple instructions at the same time on various data instances. In order to do this, they instantiate many tasks and rapidly switch between them. But CPUs have small quantities of registers per core, so

they have to switch them in and out to main memory every time they change the thread executing. This is a very time-consuming operation. For this reason programmers, when designing software that runs on CPUs, rather think in terms of a more coarse task parallelism, with only few threads instantiated that can execute quite complex tasks, in order to exploit the presence of multiple processing cores but minimising the overhead due to context switches.

In the GPU domain, we have to think the opposite, they are designed to execute a large number of simpler tasks. Moreover, they don't have small quantities of registers like CPUs, but as we said in Section 5.1.1 there is a relatively large chunk of the fast shared memory called *register file* in which more version of registers used by each core can be stored at once. Thus in order to perform a context switch is enough to put a register selector that switches between different registers version in the file, without the need to actually switch in and out registers from memory. This operation is orders of magnitude faster than traditional CPUs context switches.

This is a very important feature to keep present while designing programs. Both CPUs and GPUs have to handle stalls, generally caused by I/O operations and memory fetches. When this happens in a CPU it solves this by context switching to another thread, provided that there are enough pending task. As the number of tasks increase, considering that scheduling policies of CPUs are often based on equally dividing time between threads, the time overhead of context switches compared to the actual execution time of the tasks leads efficiency to decrease quite rapidly.

Same behaviour is adopted by GPUs to handle stalls, but, due to quite simpler nature of threads with respect to CPU ones, and the rapidity with which GPUs can perform context switches, they need thousands of thread to effectively keep the hardware busy and work efficiently. Every time it encounters a memory fetch or another blocking instruction, the GPU performs a context switch and stay busy doing some other task. This operation is performed many more times than in a CPU, and this is one of the key aspect that makes GPUs so powerful.

The task execution model of a GPU is a generalisation of SIMD, Single Program - Multiple Data (SPMD). The main difference is that SPs are grouped into blocks that execute the *same program* in a lock-step basis. This means that every instruction in the program queue is fetched simultaneously to each SPs of each

block that execute it on different data. A traditional CPU would fetch different instruction flows to each of its cores. This leads to a require a 1/N memory bandwidth to fetch instructions from memory, where N is the number of SPs belonging to the same block.

5.4 Programming with CUDA

CUDA issues threads on the hardware in large groups, following SPMD model. In the CUDA context, the base unit of such groups, is composed by 32 threads, namely a *warp*. A group of 16 threads is called *half warp*. GPU hardware includes one or more warp dispatcher that is in charge to issue warps on Streaming Processors, and actually it is not possible to run threads in different numbers than multiples of a warp. If you instantiate 50 threads, the GPU will still and issue commands to two warps, 64 threads will be running and you will just waste 14 of them. In order to make easier to discuss how in practice CUDA works in terms of thread numbers and arrangement, let us present a practical example.

```
1 void func(){
2   int i, *a, *b, *c;
3   for(i = 0; i < 2048; i++)
4     a[i] = b[i] * c[i]
5   return;
6 }
```

Source 5.1: The function performs the product between arrays `b` and `c`, storing the result in `a`.

A serial execution of the function illustrated in Source 5.1, would sequentially repeat 2048 times the multiplication between i-th cell of b and c, storing result in i-th cell of a. Now we want to parallelise this piece of code in CUDA. Given that each iteration of the loop is independent by any other, the loop execution can be split and distributed to 2048 separate and parallel threads that perform only one multiplication. This is what the informatics call an *embarrassingly parallel* problem. In CUDA you can implement this by writing a *kernel function*, which is a particular class of functions defined into CUDA C/C++ environment that can be executed only on the GPU by bunch of threads.

5.4.1 Kernel functions and thread grid

In order to define jobs that we want to execute on the GPU, we must write a *kernel function*. A call to a kernel function can occur in the CPU code when needed, and it will instantiate a *kernel* on the GPU hardware.

Nvidia also defined CUDA *streams*, a sequences of tasks (i.e. kernels) that are executed on the GPU in their issue order, in an asynchronous fashion, while CPU continues its execution flow. Kernels are always issued inside a stream, if none is specified, the default one will be used. Starting from version 7 of the CUDA API, a default stream is created for each CPU thread when a GPU device is initialised in the program. More streams can be created through `cudaStreamCreate(...)`. Issuing kernels into different CUDA streams allows to execute them concurrently on the GPU, and multiple kernels issued in different streams may be also executed out of order with respect to each other.

Kernel functions can be declared in the source more or less like normal C functions, but with some limitations, as illustrated in Source 5.2: they must return `void`, they can only access device memory and the keyword `__global__` must be placed before its declaration.

```
__global__ void kernel_func(const int *a,
                    const int *b,
                    const int *c){

  const unsigned int idx = threadIdx.x + (blockIdx.x *
                     blockDim.x);
  a[idx] = b[idx] * c[idx]
  return;
}
```

Source 5.2: A kernel function declaration.

CUDA defines a particular extension to C language notation in order to invoke a kernel function inside the CPU code. After the name of the function, and before specifying the function arguments, other parameters must be placed. The notation is `func_name<<< Dg, Db, Ns >>>(args)`.

Values `Dg`, `Db` and `Ns` are respectively two `dim3` variables and an unsigned integer. Starting from the last one, `Ns`, it is the identifier of the stream in which we want to issue the kernel. It is an optional parameter and, if not present, the default stream will be automatically selected.

The other two parameters allows us to specify each dimension of the grid of threads that we want to create, i.e. how many threads will be issued, and how they will be arranged in the GPU. In fact, `dim3` type is based on `uint3` type, in which three different unsigned integers values can be stored and accessed using `var.x`, `var.y` and `var.z` notation.

More precisely, `Dg` indicates the dimension of the grid, expressed in terms of the number of thread blocks for each Cartesian axe, and `Db` indicates the size of each thread block in terms of number of threads for each Cartesian axe.

The grid organisation of threads is one of the most interesting characteristics for the CUDA environment, and we are now going to explain it better.

Thread grids and thread blocks

When kernel functions are issued to the GPU, a grid of threads is created. A thread grid is nothing but large group of threads that execute the same kernel function. Its dimension is specified by the programmer at the moment of calling a kernel function through the first of the additional parameters introduced by Nvidia to the C language extension. As illustrated in Table 5.1, since compute capability 2.0 is it possible to create both 2D and 3D grids of threads.

Threads that runs in the same grid are additionally grouped in blocks in order to simplify data mapping, scheduling e processing of tasks. The *thread block* is an extremely important entity in the CUDA environment, since lots of different aspects are defined on a thread block basis, such as the shared memory described in Section 5.2.

Various limitations are present on the number of thread we can put in a single block and their disposition, depending on the device compute capability level, as you can see in Table 5.2.

Considering, a device belonging to compute capability 5.0, any one of the following limit must not be exceeded: no more than 1024 thread can be put in a single block, no more than 32 block can be instantiated on a single streaming multi-processor, and, generally, no more than 2048 thread can run on a SM. For example, is possible to instantiate 8 blocks of 256 threads, 16 blocks of 128 threads or 2 blocks of 1024 threads, but is not possible to instantiate only one block of 2048 thread, or 32 blocks of 512 threads or even 1024 blocks containing only two

Features	Compute capability (version)		
	2.x	3.x	5.x/6.x/7.x
Max num of threads per SM	1536	2048	
Max num of threads per Block	1024		
Max num of blocks per SM	8	16	32
Number of registers per SM	32K	64K-128K	64K
Max num of registers per thread	63	255	

Table 5.2: Details of maximum thread numbers per block and per streaming multi-processor, with the progress of compute capability levels.

threads each one.

Thread warps

In addition to size limitations we just discussed, an aspect that must be taken into account, is that the basic unit of execution in the CUDA environment is the *thread warp*. A warp is a group of 32 threads that are actually executed simultaneously on the same streaming processor. It is not possible to execute less than a warp of threads, thus if, for example, we have a job for only 20 threads, instructions will be anyway dispatched to 32 threads, but 12 of them will be wasted. Programmers have to consider in the code that not enough data may be present for each thread, and perform proper checks in order to avoid conflicts or memory failures. For this reason, where the data make it is possible, it is always suggested to instantiate blocks composed by multiples of 32 threads, so that no thread is wasted.

Get the thread position at runtime

As we said every threads belonging to a kernel execute exactly the same code on different data, following the SPMD execution model. But, if the kernel function code is the same for each thread, we need some parameter that can tell to each thread what data it must select and work with. To solve this problem, in CUDA threads are provided with positioning indices inside the grid in which they are issued.

Inside each block, threads are numbered following each Cartesian axe. Indices can be read by each thread by four `dim3` variables defined in CUDA, called `threadIdx`, `blockIdx`, `blockDim` and `gridDim`. The first one indicates the index of the current thread inside its block, the second one indicates the index of the block to which the current thread belongs with respect to the thread grid; the third variable specifies the size of each block and the last one describes the three dimensions of the grid. In order to get univocal IDs for each thread, these two values can be combined. Supposing to instantiate a kernel with only one thread block, only threadIdx could be used, but this is not the general situation. The examples in Source **??** shows two possible cases, one with one-dimensional grid, and one with bi-dimensional grid.

```
__global__ void kernel_func_1( <params>, ...)  {
  /* the case of one-dimensional blocks */
  int thread_idx = (blockIdx.x * blockDim.x) + threadIdx.x;
}

__global__ void kernel_func_2( <params>, ...)  {
  int idx = (blockIdx.x * (blockDim.x) ) + threadIdx.x;
  int idy = (blockIdx.y * blockDim.y ) + threadIdx.y;
  int thread_id = idx + idy * (blockDim.x) * (gridDim.x);
}
```

Source 5.3: Grids, blocks and threads indices.

In the first function, the univocal ID of the running thread is computed by taking into account the index of the thread inside its block, and the `x` dimension of the block. In the second function, both the `x` and the `y` dimensions of the block are considered, as well as the `x` dimension of the grid;

Once each thread has computed its own univocal index, they can use it to separately access to respective portion of data.

Passing parameters to kernel functions

Since the GPU and CPU make use of separate address spaces, it is forbidden to dereference device's pointers in the CPU, or the reverse: they can only communicate through specific functions via system buses that link CPU main memory to device global memory.

Names of those functions are taken from the C-language semantics, by adding prefix `cuda`. So `cudaMalloc` allocates memory on the device global memory,

cudaMemcpy copy memory from CPU main memory to device allocated global memory and so on.

```
1  int a[2048], b[2048], c[2048];
2  // initialise a, b and c;
3  int *d_a, *d_b, d_c; //device pointers
4  cudaMalloc((void**)&d_a, 2048 * sizeof *d_a);
5  cudaMalloc((void**)&d_b, 2048 * sizeof *d_b);
6  cudaMalloc((void**)&d_c, 2048 * sizeof *d_c);
7
8  cudaMemcpy(d_a, a, 2048 * sizeof *d_a, cudaMemcpyHostToDevice);
9  cudaMemcpy(d_b, b, 2048 * sizeof *d_b, cudaMemcpyHostToDevice);
10 cudaMemcpy(d_c, c, 2048 * sizeof *d_c, cudaMemcpyHostToDevice);
11
12 kernel_func<<< 8, 256 >>>(d_a, d_b, d_c);
13 // cudaDeviceSynchronize();
14 cudaMemcpy(d_a, a, 2048 * sizeof *d_a, cudaMemcpyDeviceToHost);
15 cudaMemcpy(d_b, b, 2048 * sizeof *d_b, cudaMemcpyDeviceToHost);
16 cudaMemcpy(d_c, c, 2048 * sizeof *d_c, cudaMemcpyDeviceToHost);
```

Source 5.4: Sequence of functions needed to launch kernels with parameters.

Let us continue the example of Source 5.1 and 5.2: in Source 5.4, a and b are the arrays we want to sum and c is the desired result array. As we said, we cannot pass directly these arrays to the kernel, because they reside into the CPU main memory. We first must allocate enough space for them into the GPU global memory (lines 4-6), then we have to copy there the arrays. cudaMemcpy will store in d_a, d_b and d_c the actual pointers to device global memory where arrays a, b and c have been copied. Now we can pass these pointers to the kernel function.

In order to synchronise CPU and GPU so that the CPU waits until the GPU finishes the execution of the kernel, a specific function would be necessary, cudaDeviceSynchronize (line 13). However, an implicit synchronisation mechanism is already included in cudaMemcpy, which wait that previously launched jobs on the GPU are terminated before coping the memory. Thus, in our case, explicit synchronisation is not needed. In lines 14-16, values of arrays d_a, d_b and d_c are copied back into a, b and c, and now we can access the desired result. Note that same function cudaMemcpy is used to copy memory from and to the device global memory, by using parameters cudaMemcpyHostToDevice and cudaMemcpyDeviceToHost.

that we wan in order to pass parameters to the GPU you have to allocate new memory on the device global memory through cudaMalloc ad copy there what

you need through `cudaMemcpy` using the parameter `cudaMemcpyHostToDevice`.

5.4.2 Best practices

Just by following the programming guide, using the correct cuda functions in the correct place, we should be able to obtain working CUDA code. We can compile it using Nvidia `nvcc` compiler and then execute it. It can happen that the impressive performances that we would expect to observe, actually turn out to be rather disappointing. Of course causes of bad performances can be various, such as errors in our program logic, or hardware optimisation issues and so on.

Anyway, there are two important rules of CUDA architecture that programmers must care about in order to obtain good performances for their code, i.e. the necessity to maximise the memory access coalescence between threads, and, at the same time, to minimise the branch divergence of the code.

We will see, in Chapter 7, that these two critical aspects are not trivially feasible for the McSplit algorithm, and this is the main reason why the performances of the algorithm version that we are going to present in the next chapters are not satisfying in the CUDA environment, and the implementation itself cannot be considered a success.

Data locality and memory access coalescence

Fetching data from the global memory of the device is a very time-consuming operation. When a thread needs to read a variable that resides in global or local memory, it has to block for hundreds of clock cycles, waiting for data coming from the memory. One of the strength points of GPUs is that they can perform very fast context switches, and hide waiting phases by executing other threads. This approach can lead to satisfying performances, of course provided that memory fetches are not too many.

Considering that all the threads execute the same code, if a variable is needed from the local or global memory, hundreds or thousands of memory fetches at the same time will be performed. Such situation actually would occur every time threads access the global or local memory. This is one of the main reason why CUDA threads are grouped into warps. In fact, the device coalesces global memory loads and stores issued by threads of a warp into as few transactions as

possible to minimise the number processors stall cycles.

Anyway, this approach is possible only if the needed data, for threads in the same warp, are actually physically close also in the memory. Otherwise separate fetches will be necessary for each thread asking for sparse data in the memory, and performance will drop down significantly.

Instruction parallelism and branch divergence

In parallel with memory access coalescence, another important feature that come with warp grouping of thread, is a direct consequence of the fact threads execute the same code with different data. This system can perform pretty well, provided that actually each thread needs to execute the same instruction on any circumstance. Obviously this is not always the case: sometimes, some instruction must be executed or not depending on the data on which the thread is working, i.e. the code can present branches.

Keeping valid the general rule that every threads execute the same instructions at each clock cycle, when branches occurs, and not every thread needs to take the same branch, we obtain what we call a *branch divergence*. In such situation, executable code of the kernel function would include both the branches one after the other, as it happens for CPU code. When the branch is executed, let as assume the first half of the threads in the warp will execute the first branch, and the remaining thread will simply stall. When the first branch is finished, the second half of the threads will execute the second branch, while the first one stall and waits. Thus when branches are encountered in the code, we can assume that both branches will be executed in sequence by some portion of the warps, wasting computing time for the remaining part. The more branches are present in the code, and the longer they are, and the more the performances will be affected by branch divergence.

5.4.3 CUDA Dynamic Parallelism

CUDA Dynamic Parallelism was introduced in 2014 within compute capability 5.0 with retro-compatibility down to 3.5 devices. It offers the possibility to create, execute and synchronise with new kernels directly from the the GPU, without any participation from the CPU.

This feature is useful, for example, when data with different granularity are distributed in the space, we can for example instantiate a main kernel with fewer smaller blocks to cover the whole data space. Then we can make the threads themselves to delegate eventual area of data space, where data are closer, to child kernels composed by more threads, that process it and then return to the parent thread.

Suppose, for instance, you have to analyse fluid dynamics data that represent the output of hundreds of thermal sensors aimed to track warm oceanic currents. It is not useful to finely analyse data far from the current, while it can be interesting to focus on the borders of the current. Well, in this case a kernel with coarse block subdivision can be launched to analyse the whole data space, and then in proximity to sensors that registered thermal variations, threads from the coarse kernel can also instantiate kernel functions that analyse finer data.

Of course it is also possible to create recursive kernel functions, so that the same task may be applied with different granularity to portions of data, but some limitation are present.

Unfortunately, even in high level GPUs, where hardware resources are abundant, if we consider them in per thread basis, they are absolutely not enough to sustain a complete function call stack with a large number of entries, as it happens in the CPU. On the contrary, in the CUDA environment, kernel recursions are limited up to 24 levels of depth. If this number may seem too small, remember that every threads running in the GPU is supposed to have a limited number of registers and local memory. Those same resources should be also shared with every child recursion starting from that thread. It is clear that in most of cases, hardware resources would run out much before reaching 24 levels of depth. Thus, even if CUDA dynamic parallelism can be very useful in lots of application fields, such as fluid dynamics analysis, actually it is not applicable in order to implement properly recursive functions.

Chapter 6

A McSplit CUDA implementation

We have discussed in Chapter 4 about currently available implementations of the McSplit algorithm to solve the maximum common subgraph problem (MCS), and we presented the CUDA environment in the previous chapter. Now we are ready to continue the dissertation with some consideration about the available implementations, ad by introducing our attempts to modify them in order to make them suitable to run on Nvidia GPUs.

Several different approaches has been tried, during our work, but not all of them are relevant considering the final result. Thus, here we will present only few different implementations that are useful to retracing the path we followed to produce our CUDA implementation.

Making a completely running version of the algorithm, exploiting every aspect of the CUDA environment in order to immediately have a well performing code would have been very difficult. Thus first we started by analysing in depth the existing implementations, studying which element could be saved and which other would have been incompatible with CUDA. Finally we created two new CPU implementations, one sequential and one parallel, that actually emulates the final implementation we produced for the CUDA environment.

Removing recursion

As we widely discussed in previous chapters, the McSplit algorithm is inherently recursive, but even using Dynamic Parallelism, CUDA does not support more than 24 in-depth recursions. The maximum recursion depth reached by the Mc-

Split algorithm is equal to the size of the final solution, and thus may vary from one up to the degree of the smaller input graph. Supposing to leave the recursion untouched, we could implement in CUDA an algorithm suitable for graphs with at maximum 24 nodes. Of course this would be practically useless, given that for graphs with such order, the CPU-only implementation still perform pretty well even in a total-sequential flow. For this reason, one of first step necessary to adapt the algorithm to CUDA is to modify the execution structure in order to formally remove recursion.

If you understood the structure of the `solve` function presented in Section 4.1, you could imagine that removing recursion from there actually is not so simple.

In fact, a `for` loop is present, and new recursion calls are performed in each iteration of the loop, plus one more call outside the loop. The first naive attempt we made, was just to wrap all the `solve` function argument in a structure, and then arrange it in a stack. With a `while` loop and such stack, recursion can be obtained by imitating the actual behaviour of CPUs: each iteration of the `while` loop correspond to an execution of the `solve` function, it pop a set of arguments from the stack, and execute more or less the same code as `solve` function body. Inside, at each iteration of the `for` loop, instead of a recursive call to `solve` a new set of arguments is wrapped in a structure and pushed on the stack. The following `while` iteration will pop it and execute with such arguments, and so on. In order to maintain the same recursion order than the standard function, first an attempt to leave vertex from the first graph unmatched is done, then the `for` loop is executed backwards. Thus the arguments are pushed in the stack in reverse order, and later popped in the same order than a standard recursion.

Reducing memory usage

This approach, besides it is not very sophisticated, still remains practically unfeasible on GPUs. Unlike standard recursive flow, here, at each level we put on the stack every arguments for each virtual recursion of the `for` loop. This leads in just few depth levels to have great amount of memory wasted storing arguments for virtual recursions we will actually explore much later in the algorithm execution. The scaling of this issue is exponential in the recursion depth, but also unpredictable. Remember that per thread memory on GPU is quite limited.

Moreover, with the proceeding of the algorithm, arguments sets present on the stack correspond to branches for which we can assume very varying depth, because they are been created at different depth levels. Trying to assign large portion of the stack to the GPU in order solve in parallel each argument set would not be a smart solution, because it would lead to an extremely unbalanced workload on each GPU thread. For this reason we had to find another approach able to greatly decrease the algorithm memory usage, while maintaining the absence of recursion.

6.1 First implementation: naive labels re-computation

A first naive attempt we did was to sacrifice some algorithm smart computation in order to simplify the code structure. This of course involves a huge drop of performances considering a sequential execution, but we was hoping that more parallelism would be achieved with this approach, actually balancing the sequential performance drop with large amount of possible parallel work.

Thus, in this version we got rid of bidomains data structures, together with left and right arrays needed for their functioning.

Also the stack in which previously we put all the arguments necessary for a standard `solve` function execution has been completely modified. Now each stack entry only consists into a pair of integers values. Such values correspond to possible mappings, for each position in the solution, between vertices from the first and second graph. The position in the solution is indicated by a variable that is increased and decreased using special reserved entries in the stack: $\langle 0, -1 \rangle$ and $\langle -1, 0 \rangle$. The first one is needed to move the computation on one position left in the solution array, and the second one to move the computation on one position right in the solution array.

For better understand this concept, a small practical example is presented in Figure 6.1 with two hypothetical graphs of order 5. Suppose that at the beginning of the algorithm, any combination of one vertex from the first graph and one from the second graph is possible, the stack will contain each of those pairs, from $\langle 4, 4 \rangle$ down to $\langle 0, 0 \rangle$, in reverse order, so they are popped out from the stack in the same order than the standard algorithm. Remember that any value in the stack has effect only when it is popped out from the stack and not when it is pushed

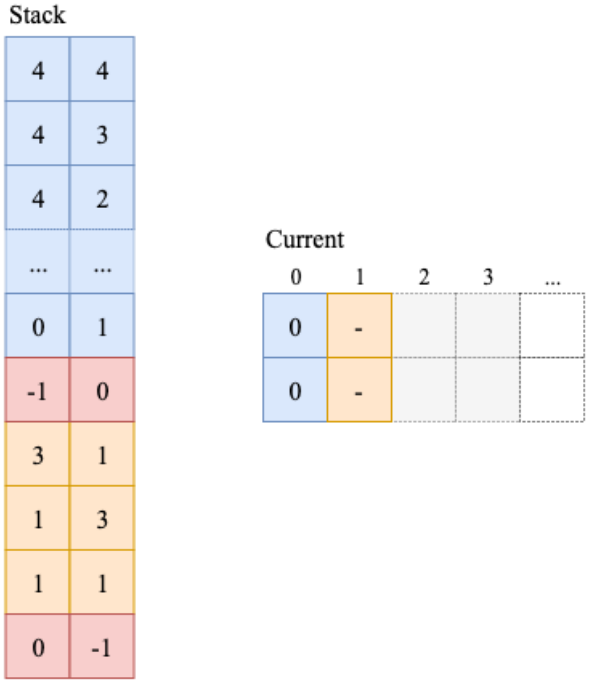


Figure 6.1: Stack and current solution during naive labelling approach

in.

In the example in figure, the initial possible values are marked in blue, and pair $\langle 0, 0 \rangle$ has already been popped out and put in the solution array. Now, considering pairs already in the solution, a function computes label classes for each remaining vertex from both the input graphs, and push in the stack all combinations of pairs composed by vertices with matching labels (identified in orange in Figure 6.1).

Before and after pushing such pairs on the stack, the two reserved sequences are pushed, so that $\langle -1, 0 \rangle$ come before and $\langle 0, -1 \rangle$ at the end. In this way the next pair that is popped out is $\langle 0, -1 \rangle$, the current solution position is increased from 0 to 1, and the algorithm starts popping out orange pairs that are put in the second position. For each of them, more vertices will be pushed in the stack for the third position and so on. When every orange pair has been popped, $\langle -1, 0 \rangle$ is extracted, and the solution position returns to 0, pair $\langle 0, 1 \rangle$ selected and the algorithm continues until every blue pairs have been popped. Every time a better solution is found, it is stored in the incumbent solution, similarly to the standard

algorithm.

As you may have noticed, the just described stack implementation is not really feasible. At the beginning of the algorithm any pair of vertices can be selected to be the first one, thus the first entries of the stack are occupied by all the possible combination of vertices from the two input graphs. The number of such pairs is the product of the degrees of the two input graphs, so it quickly increases with the increasing of the input graphs degree.

For each possible starting pair, the algorithm begins and perform its computations. Thus an easy optimisation would be to put the algorithm, inside two nested `for` loops, that iterate on every nodes of the input graphs respectively, and for each inner iteration only one of the possible starting pair is pushed in the stack for the algorithm.

In such a way the resulting algorithm remains the same, but we don't have to store in the stack every combination of vertices since the beginning.

Downsides

A fundamental aspect for any branch and bound algorithm that attempts to solve the maximum common subgraph problem, is that it must be able to efficiently prune the most of the branches in order to not waste computing resources on paths that don't lead to better solutions.

Unfortunately, this naive approach to labels re-computations, doesn't allow to efficiently calculate a bound for each new branch. Indeed, every time a new pair is selected and put in the solution, we have to compute the bound of the new branch by going re-computing the label for each of other vertices, and counting how many vertices with the same label there are in the two graphs, then sum the minimum value for each label class, applying in practical the bound formula (3.1).

Moreover, the necessity to store separately each vertex pair belonging to the same label class, instead of grouping them into a bidomain that is able to contain them all, leads to a quickly increasing stack size with the progress of the algorithm, and this may fast become unfeasible.

The bound calculation operation, applied each time a new pair is popped from the stack, together with the memory management aspects, leads to massive

slowdowns that actually goes against the purpose for which this approach has been created, and make questionable its use in order to solve the problem in reasonable time.

6.2 New bidomains stack

The next step of our work was to keep what of good there was in previous version, i.e. the structural simplicity and the recursion simulation with a stack composed by small entries and try to improve it by reinserting some advanced feature from the standard algorithm, opportunely modified for this purpose. The main structure with a stack and a while loop has been maintained, but the internal logic has been heavily modified.

An important need that had to be satisfied was being able to group again vertices in label-classes, because storing all of them separately, at each research level, were completely unfeasible. Another fundamental aspect of course was to keep the algorithm iterative, because deep recursion is not feasible in CUDA environment.

In order to solve this first two problem the stack has been redesigned to store bidomains data, in an optimised form. New items are pushed in it every time a new set of bidomains is created, but then they are popped out only after every pairs of vertices contained in such bidomains has been selected and explored.

The stack has became a matrix with eight columns and variable number of rows. Each column is one `unsigned char` size, so that each stack entry is eight byte long, similarly to one described in the previous section, that contained two integer values.

The purpose of this new algorithm is to emulate the most possible the behaviour of the standard McSplit algorithm, but at the same time to keep the code the most possible compact and simple.

Current solution and incumbent structures has been replaced by `unsigned char` matrices of two rows and variable columns. For sake of simplicity and code readability, in our purposes we used over-allocated static matrices for bidomains stack and solutions.

The use of `unsigned char` for values storing may seem a too narrow restriction, but actually numbers that will be stored in such variables are at most as

large than the degree of the larger input graph, thus we can handle graphs with up to 255 vertices (value 255 is reserved for the algorithm utility). Considering the high complexity of the problem, and the large amount of time necessary to solve it with graphs with just 50 nodes, we believe that `unsigned char` is by far large enough to work with, at least in an academic context. Machines able to deal with maximum common subgraph problem and graphs larger than 255 vertices in reasonable time should be also provided with much more memory than normal consumers personal computers, so the same algorithm could be adjusted to work with unsigned integers too. Moreover, stack entries of eight bytes allow a better memory alignment of the stack that can lead to benefits in memory performances and addresses calculation.

The set of the eight increasing integers numbers that are the indices of columns of the stack has been redefined with letters for improved readability, so in all the presented code, until the end of this chapter, the following definitions hold:

- `#define 0 L`
- `#define 1 R`
- `#define 2 LL`
- `#define 3 RL`
- `#define 4 ADJ`
- `#define 5 P`
- `#define 6 W`
- `#define 7 IRL`

where each value stands respectively for ***L**eft, **R**ight, **L**eft **L**ength, **R**ight **L**ength, **Adj**acent, **P**osition, last **W**, **I**nitial **R**ight **L**ength.*

The first five values actually correspond to the ones included in the `bidomain_t` data structure, described in Chapter 4, with the same meaning. `L` and `R` are also used to identify the two rows in current solution and incumbent matrices, such that the first row, containing vertices from the left graph has index `L`

and the second row, containing vertices from the right graph, has index `R`. You can see Figure 6.2 to better understand how these data structures are formed.

The remaining three columns in the stack are used to store values that are necessary to the execution of the algorithm. In fact, given that we cannot count on recursion properties that restore values of certain variables when backtracking, we need to explicitly store some values from previous iterations in the stack in order to restore them manually when needed.

The new algorithm working

The main logic of the new algorithm is pretty much similar to the standard algorithm one: at each iteration of the main `while` cycle, we choose a vertex from left bidomain and a vertex from right bidomain, then we put them in the solution and we compute the new bidomains depending on the vertices we just chose.

If you remember, the practical selection of a vertex is performed by swapping it with the one at the end of the bidomain and by decreasing bidomain size. When removing vertices from the right bidomain, we also must be able to put them back again, because they must be matched with every possible vertex from the left bidomain. In the recursive algorithm, this is simply done exploiting recursion properties. When backtracking to lower recursion levels, deeper bidomains, that have been shortened removing vertices, are destroyed, the old ones has not been touched by deeper recursions, and the algorithm can continue with consistent data structures. Instead, due to the logic of the `McSplit` algorithm, for vertices in the left bidomain the management is simpler, because once we have removed them from the bidomain, we never need to restore them.

Now that recursion is removed, however, we have lost its *automatic restoring* property: when we select a vertex from the right bidomain and we decrease its size, the starting size must be stored somewhere in order to be able to restore it when every right vertices has been matched with the selected left vertex. The values of the last column of the stack have precisely this purpose, every time a new bidomain is creates, is there stored the initial value of `RL`, for future utility.

In a similar way, after the recursive call placed inside the `for` cycle of the standard recursive `solve` function returns, we continue the cycle as if nothing happened, thus we know which right vertex we just selected and we can correctly

Bidomains Stack

	L	R	LL	LR	ADJ	P	W	IRL
0	0	0	9	9	0	0	0	10
1	0	0	4	5	1	1	1	6
2	5	6	4	3	0	1	255	3
3	0	0	2	1	1	2	5	2
4	3	2	1	3	0	2	255	3
5	-	-	-	-	1	2	255	-
6	-	-	-	-	0	2	255	-
7	1	0	1	1	0	3	255	1

Current Solution

	0	1	2	3
L	0	2	4	...
R	0	1	5	...

Figure 6.2: Example of the possible state of bidomains stack and current solution matrices after three iterations of the algorithm

pick the next one for the next recursion.

Since we now don't have such recursive call for each iteration, we designed the second-last element in the stack for keeping track every time of which right vertex has been just selected, in order to correctly pick the next one.

Finally, any bidomains set were valid only for the execution of a certain recursion level: higher levels were handled with new bidomains created by `filter_domains` function, and lower levels were handled by old bidomains. Now all bidomains coexist on the same stack, and we have to separate them depending on the solution level for which they are valid. Thus we used the last free value in the stack to store the position in the solution for which the bidomain is valid. This value allows us to separate bidomains, and correctly calculate bounds and selecting the best bidomain at each new iteration of the algorithm.

A practical example

In Figure 6.2 is presented the state of the stack as it would be at the end of the same example we already followed in Figure 3.4, so we can now better understand what we just illustrated in previous paragraphs. Same colour code has been used to provide a readable parallelism between the two examples.

Initially, since both `n0` and `n1` are 10, the first bidomain pushed into the stack should be `[0, 0, 10, 10, 0, 0, 255, 10]`. The first five numbers are the same of the standard bidomain structure, then in `P` position there is 0, as the first bidomain is valid for the first position of the solution, in position `W` there is 255, meaning that no vertex has been selected from right bidomain yet, and finally, in position `IRL` the initial value of `RL` is stored, that is 10.

Since first few iterations of the algorithm have already been done in this example, the first pair ⟨`0,0`⟩ for position `0` has already been formed, pairing `0` from left bidomain and `0` from right bidomain. Thus we can see that the *blue* bidomain, has values `[0,0,9,9,0]` (both left and right length of the bidomain were decreased when first values have been picked), with position `0`, last vertex chosen from right bidomain is `0` and the initial value of `RL` was `10`.

When the algorithm will have explored every solution containing ⟨`0,0`⟩ and thus it will have popped everything else from the stack and will be returned down to first position, it will be able to correctly select the second lowest value after `0` from the right bidomain, that is `1` and create pair ⟨`0,1`⟩, from which continue the execution.

From pair ⟨`0,0`⟩ and bidomain in position 0, two new bidomains are created and pushed on the stack in position 1 and 2, following the same logic of the standard algorithm. As you can see, the red bidomain still has not been considered, so the `RL` column and `IRL` column contain the same value.

Bidomain in position 2 is selected for position 1 of the solution, and pair ⟨`2,1`⟩ is added to the mapping. Four more bidomains with `P=2` are obtained by splitting the ones available with `P=1` and considering the current solution [⟨`0,0`⟩, ⟨`2,1`⟩]. The last one in the stack is picked and pair ⟨`4,5`⟩ is put in the solution and so on and so forth.

For each branch the bound upper bound is calculated in the same way of previous versions, by applying the bound formula (3.1) reading backwards the stack as long as bidomains with same `P` value than the last one are found. When a branch hits the bound, backtrack is performed by removing the whole set of bidomain with the same `P` from the last positions of the stack. The `while` cycle restart and the same bidomain of the previous iteration is selected.

Every time we have to pick new values from the right bidomain, we use the value stored in position `W` in order to select the correct subsequent vertex, instead

```
add_bidomain(domains, &bd_pos, 0, 0, n0, n1, 0, 0);
while (bd_pos > 0) {
    bd = &domains[bd_pos - 1][L];
    if (calc_bound(...) <= *inc_pos ||
                (bd[LL] == 0 && bd[RL] == bd[IRL])) {
        bd_pos--;
    } else {
        v = select_next_v(left, bd);
        if ((bd[W] = select_next_w(right, bd)) != UCHAR_MAX) {
            w = right[bd[R] + bd[W]];
            //... swap w with vertex at the end of domain
            bd[W] = w;

            cur[bd[P]][L] = v;
            cur[bd[P]][R] = w;
            update_incumbent(...);
            generate_next_domains(...);
        }
    }
}
```

Source 6.1: The core cycle of the new iterative version of McSplit algorithm.

the vertex of the left bidomain should remain the same of the last pair previously created from the bidomain. In order to select again the one that previously we moved out of the bidomain, the first vertex past the end of the left domain is picked.

When every vertices of the right bidomain has been paired with the same vertex from the left bidomain, and thus we need to pick a new vertex from left bidomain and pair it again with every vertices of the right bidomain, we will be in a situation in which the corresponding stack entry has $LL \neq 0$ and $RL = 0$. We restore the initial value of `RL` reading it from `IRL`, select a new vertex from left bidomain and normally continue the execution. In such a way, every time a bidomain with `RL == IRL` is selected from the stack, the algorithm knows it has to select a new vertex from the left bidomain and not to going to pick it from the first position past the end od the domain.

Finally, when a bidomain is selected and $LL = 0$ and $RL == IRL$, this means that every possible pair has already been explored, the bidomain is empty and thus popped out from the stack.

Source 6.1 illustrates the core `while` cycle we just discussed. It has been de-

signed to be the most compact possible in the main structure and variables utilisation. Thus, for example column `W` of the stack is also used during the iteration as a temporary variable where it is stored the index in the right array of the new value that must be put in the solution pair. The full code of this new iterative version is reported in Appendix A and C.

Bidomains selection

Following this approach, every time a bidomain has been completely explored, we remove it from the stack and we pick the immediately following one, as intended in the functioning of a stack data structure. Doing so, we lose one important optimisation featured by the original McSplit algorithm.

Indeed, the algorithm just presented, without any modifications, performs significantly worse than the original on any instance with graphs larger than 15 nodes. This performance drop is in large part due to the lack of a proper heuristic approach to choose which bidomain select each time, and just by picking the first one on the stack. However it is pretty easy to implement function an equivalent to `select_bidomain` from the original implementation. The new function just goes backwards in the stack, considering the bidomains with the same `P` value than the one on the top of the stack, and compute the best domains by applying the same heuristic of `select_bidomain`. Then it exchange the best bidomain found with the one in the top of the stack. After this operation the algorithm continues normally.

6.3 Launching the new version in CUDA environment

We just introduced a new iterative implementation of the McSplit main function that is provided with very compact code, much less memory usage than the original implementation and no recursive calls in its body.

What we wanted to obtain from here was a function suitable to be executed in a CUDA kernel function, in order to exploit huge parallel computing power of the GPU to balance the sequential performance drop and eventually gain something in terms of computing time. As we will explain in the next chapter, this did not

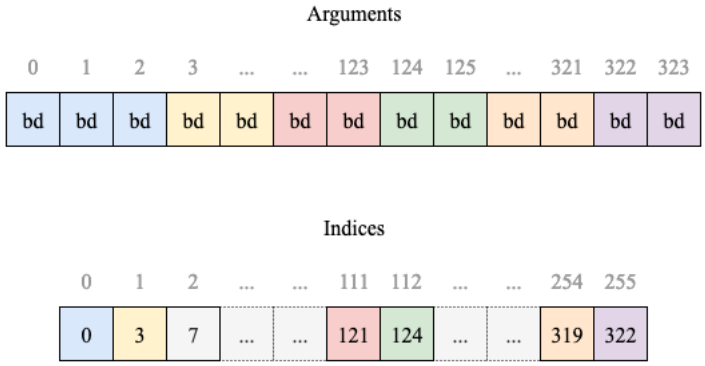


Figure 6.3: Example of the arrays containing thread arguments and the start and end indices for each thread.

happen, but it is still interesting to analyse the results and take note of what it still can be done in order to improve performances.

Given that a fundamental aspect of GPU computing is that many threads have to compute something on multiple data instances, but the maximum common subgraph problem does not start with such great amount of data, the idea here is to allow the CPU to start the algorithm as a normal sequential execution, then when a certain depth level has been reached[1], for instance the fifth one, instead of pushing new bidomains on the stack, it uses them to fill a separate array of bidomains.

When enough bidomains has been put in the separate array to fulfil the need of large amount of data, the array is copied on the GPU and a kernel is launched so that parallel threads can process lots of branches in parallel. Each GPU thread must face with the whole bunch of bidomains coming from the same branch, in order to correctly calculate bounds and next level domains. Thus a second data structure is introduced storing the start indices in the arguments array for each thread. This allows each GPU thread to pick the correct amount of bidomains from the arguments list (of course it starts from the start index assigned to it and it stops at the start index assigned to the following thread, or at the end of the arguments), you can visualise this concept in Figure 6.3.

This approach has been first used to produce a CPU multi-thread version of

[1] since no recursion is present, but it is simulated, the term depth here is intended as a virtual recursion depth

this code, that makes use of a smaller thread pool, compared with GPU threads number, but replicates the same behaviour of using a main thread inserting at a certain level bidomains in an array and then let the thread pool to continue the algorithm in parallel on each of them.

As for the sequential implementation of this iterative approach, the full code for the final GPU version of our algorithm is reported in Appendix B and C.

As we anticipated, unfortunately, counting on parallel computational resources is not enough to overcome structural incompatibility of the McSplit algorithm and, more generally, of the maximum common subgraph problem with the GPU environment. We will better discuss main reasons of non optimal performances of the CUDA implementation while analysing tests results, in the next Chapter.

Chapter 7

Results

We are going now to present some result from testing phase of our code versions, compared to original implementations produced by [McCreesh, Prosser, and Trimble 2017]. As well as our colleagues from Glasgow did, we tested our code using the ARG database of graphs provided by [Foggia, Sansone, and Vento 2001] and [De Santo et al. 2003] from Università degli Studi di Salerno.

Tests has been performed on a machine initially configured for private gaming purposes, running Ubuntu 18.04 LTS, and provided with over-clocked Intel i7 4790k (@4.4GHz) processor, 16GB (@2133MHz) of main memory, and over-clocked Nvidia GTX 980 (@1300MHz) graphic card, with 4GB of dedicated fast memory and 2048 CUDA cores belonging to Compute Level 5.2.

Datasets description

The ARG database is composed by several class of graphs, randomly generated according to six different generation strategies with various parameters settings. The result is a huge dataset of 168 different types of graphs and a total of 166.000 different graphs, for more information on the ARG database visit `https://mivia.unisa.it/datasets/graph-database/arg-database/`.

For our purposes, we selected only a subsection of the dataset, containing graphs pairs already already prepared to have maximum common subgraph of a certain size. In particular we used `mcs10`, `mcs30`, `mcs50`, `mcs70` and `mcs90` categories, that mean graphs pairs with maximum common subgraphs corresponding to 10, 30, 50, 70 and 90 percent of the original graphs size. For each of

these categories several generation strategies are present, such as *bound-valence* graphs (bvg) or graphs generated using 2D, 3D or 4D meshes with different parameters. Starting from this subset of the ARG database we reduced again the number of instances we tested by selecting only graphs pairs with less than 40 vertices. This choice was only aimed to execute tests and collect benchmark data on different execution of the tests, and we removed larger instances in order to keep benchmark execution time low enough to not keep busy the machine for more than half a day.

The resulting dataset we obtained is composed by 2750 graphs pairs, that is small enough to make benchmarks feasible in few hours, various enough to test the performances on different graph types, easy and hard instances, and at the same time large enough to have many instances of each type on which test the implementations.

Summary of tested versions

Starting from the two main versions provided by the authors of the McSplit algorithm, which have main version identifier 0, five more code versions have been tested. In this chapter we will refer to them as follows:

- *Version 0.1:* Original code, a CPU single-thread implementation written in C++, developed by James Trimble;
- *Version 0.2:* Original code, a CPU multi-thread implementation written in C++, developed by James Trimble;
- *Version 1:* Derived code, version strictly derived from some sequential C implementation developed by James Trimble, with modification aimed to keep it more coherent with following versions;
- *Version 2:* Derived code, CPU multi-thread implementation based on version 0.2, of which this is nothing but a translation in C language, with coherent syntax;
- *Version 3:* New code, a CPU single-thread implementation that removes recursion and decreases memory usage;

- *Version 4:* New code, a CPU multi-thread implementation created as a transition from v3 to a proper CUDA implementation
- *Version 5:* New code, a CPU-GPU many-thread implementation, based on v3 and v4;

All versions from 1 to 5 have been written in C language.

Note that the not every version presented here has been developed giving much importance to performances, rather trying to explore the feasibility of new approaches in developing a maximum common subgraph algorithm in CUDA environment. For instance, as you will further see, version 4 is completely out of scale in terms of performances, compared to any other version. This is because it represent a link that explore the possibility to run the algorithm implemented in version 3 on more thread that are completely independent one of each other instead of cooperating. Indeed the same system is used on a wider scale in version 5, that still is not so much competitive, but significantly better than version 4, although they make use of the same approach.

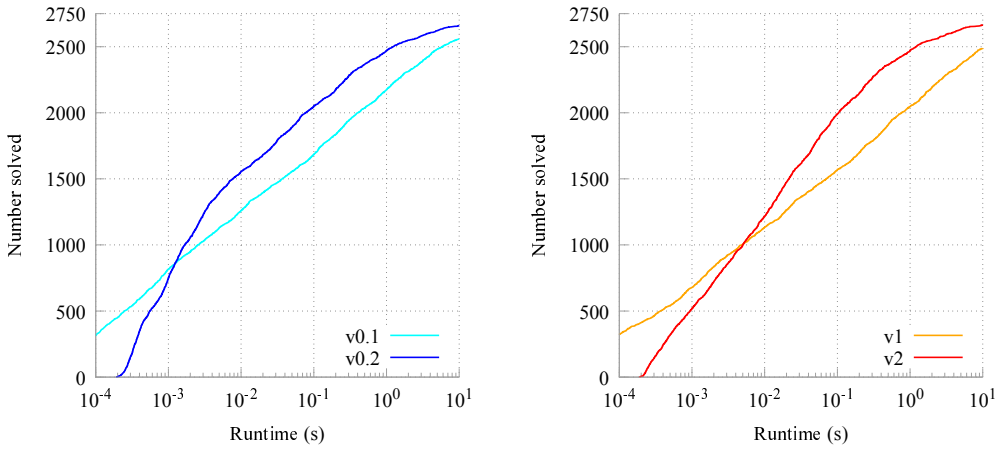
Moreover, the original versions were tuned and optimised for several month, during the drafting of a doctoral thesis, that implies much more resources and time than what we had for our work.

7.1 Analysis of the results

In the following we will compare the performances of the different code versions we just reported above.

We will start with a bunch of separate plots in which only some version is included, depending on some main features such as the flow structure or the programming language with which they are written. Doing so will allow us to explore performances more closely on versions that are similar under certain aspects, then we will compare them together in order to see advantages and disadvantages of each version.

In Figure 7.1a and Figure 7.1b are illustrated the sequential and parallel version of the original McSplit implementation, the starting point of our research, written in C++ language and C language respectively.



(a) Original C++ versions

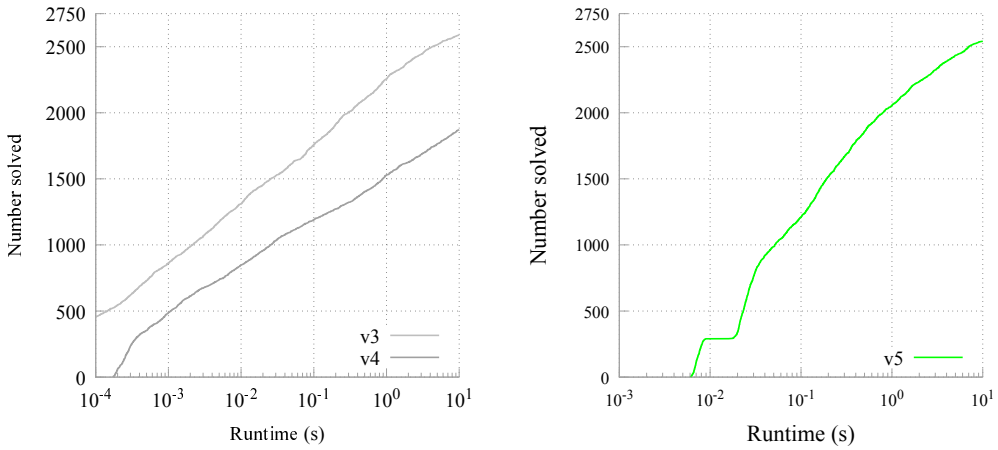
(b) Original C versions

Figure 7.1: Cumulative number of instances (y axis) solved in under a certain time (x axis).

Graphics are intended to be read as follow: given an execution time threshold on the x axis, the relative y value indicates the number of problem instances solved by the algorithm under such threshold. The abscissa axis has logarithmic scale, while the ordinates axe is standard linear. For instance, reading the C++ graphic, we can read that the parallel version solved just over 2000 instances, out of 2750, in less than 10^{-1} seconds. The sequential version took about four times the time to solve the same number of instances. Note that a slower start is present in multi-threaded versions, due to higher computing overhead for threads instantiating, thus none of the multi-thread versions can be considered faster than any sequential version to solve any small and easy instance.

Nevertheless, starting from runtimes around $10^{-3} - 10^{-2}$ seconds, parallels version quickly recover the performance gap and actually start performing significantly better for every remaining instances, with speedups up to almost an order of magnitude.

In Figure 7.2a, there are illustrated the two new versions without recursion that we introduced in the previous chapter. It is immediately clear that the sequential version (v3) performs more or less the same as the recursive corespondents. On the contrary, the multi-thread iterative version (v4) is actually the worst performing version of the group. This fact can be explained by remem-



(a) New iterative versions

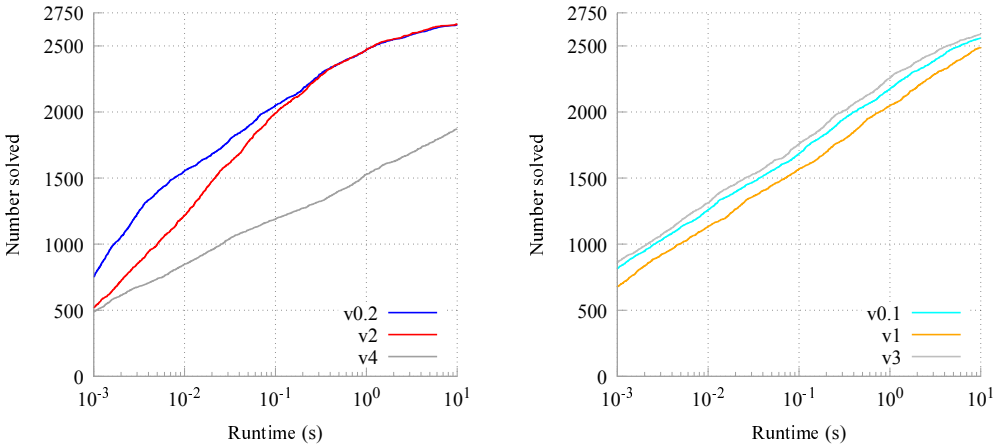
(b) CUDA version

Figure 7.2: Cumulative number of instances (y axis) solved in under a certain time (x axis).

bering that such version was only developed to explore the possibility to implement the sequential algorithm in a multi-thread but non-collaborative way. Thus threads in this implementation are independent, and performances can better scale by increasing the number of threads, what is desirable attempting a CUDA implementation.

In Figure 7.2b, the performances of the CUDA version of the iterative approach is plotted. You can see that this version is significantly slower than any other to start, this is obviously due to the high computing overhead necessary to reset and initialise the CUDA environment. You can also clearly see that the plot has a flat step in the bottom part. This phenomena is due to the fact that the algorithm runs on the CPU up to the fifth depth level of the virtual research tree and only after such level branches are delegated to GPU threads. In any instance for which the solution turns out to be very small (i.e. a maximum common subgraph with up to 5 vertices), actually the GPU is never called into question, and the computation is entirely done by the CPU. Such instances, in our database are about 260, and they are the first growing part of the plot, up to the step. Then when instances begin to have larger solutions, and thus also the GPU is used to solve them, a gap is formed due to the need of copying data to and from the GPU. For this reason instances that make use of the GPU can not run for less than

about 20 milliseconds, and this determine the gap in the graphic. From there on, performances increases much faster than any other version, actually going to almost equalise CPU parallel versions in number of timeout instances, even with a much slower start.



(a) Multi-thread summary

(b) Sequential summary

Figure 7.3: Cumulative number of instances (y axis) solved in under a certain time (x axis).

Back to results, in Figure 7.3a, is plotted a summary of the three CPU-only multi-thread versions. Here can be better observed that the iterative multi-thread version perform worse up to two orders of magnitude with respect to the original parallel versions. Moreover, it is possible to notice that the parallel C++ version performs generally better for instances of simpler and medium difficulty, while going up to harder instances, performances of the two algorithms are more or less the same, with almost negligible speedup for the C version.

A different situation is shown in Figure 7.3b, where we can see that all the three versions of the code performs more or less the same, with a gap of about half an order of magnitude between the slowest and the fastest version. It is interesting notice that for the sequential approach, the iterative approach is the fastest one.

Finally, in Figure 7.4 is plotted a larger summary of every tested version together, where overall comparison is possible.

In this graphic you can see that the original parallel versions, both C and

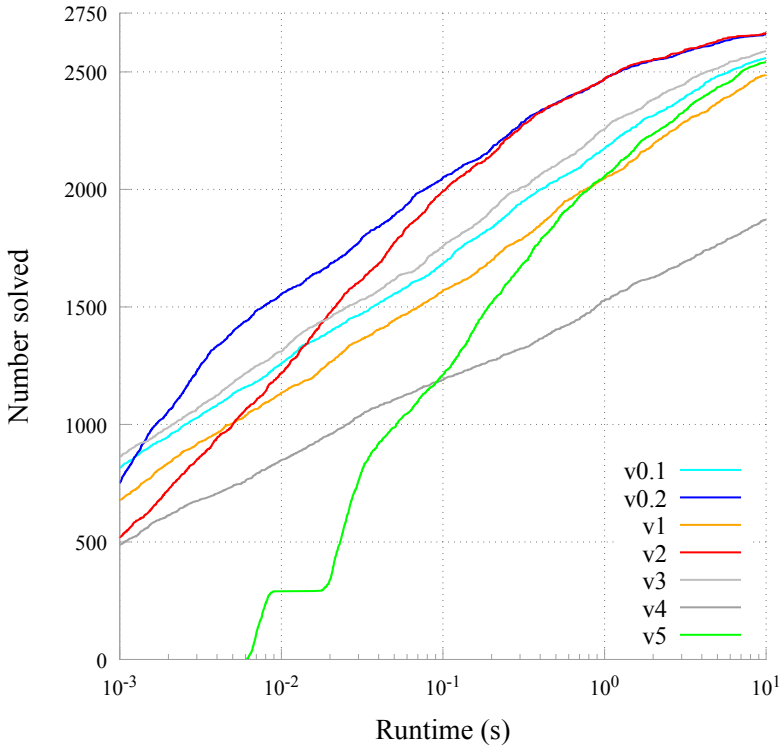


Figure 7.4: Cumulative number of instances (y axis) solved in under a certain time (x axis).

C++ ones, still are the best performing implementations. It is worth noting that the CUDA implementation, even if not competitive with the original parallel implementations, still is preferable to the simpler original C sequential version for every runtimes higher than one second. So for harder instances, CUDA can be helpful. In addition to this consideration, it can be noted that the tangent line to the curves in the timeout point, at 10 seconds, is almost horizontal for the original parallel versions, while it is more inclined for the CUDA one. This let us imagine that for even harder instances, or giving more time to solve each instance the CUDA implementation could slightly recover the performance gap between itself and the original parallel implementations. Anyway, looking at the thread it is not reasonable to imagine that the CUDA version could become faster than the original implementations for reasonably short runtimes (i.e. under 100 seconds).

Anyway, given that CUDA utilisation usually, for certain problem categories,

can lead to speedups up to several order of magnitude, it is evident that for our circumstances, the performances of our CUDA implementation definitely do not meet our expectations and our hopes.

7.2 CUDA best practice violation

Now that we have analysed result and compared the main versions discussed in this thesis, we are going to advance some hypothesis to explain the reason why the current CUDA implementation of McSplit algorithm does not perform as we hoped.

As we explained in Section 5.4.2, the two main aspect that influences the performances of a CUDA kernel are the principle of data locality and the branch divergence.

The first one needs to be maximised, so that when threads in the same warp access to a variable in their local memory, each request can be grouped in one single memory fetch of consecutive memory locations, and the the kernel takes advantage in terms of memory bandwidth.

The second aspect, the branch divergence, needs to be minimised the most as possible, since every time threads in the same warp hit a branch instruction and they take different routes, the whole warp execute both the routes sequentially, with part of the treads stalling for the first route instructions and part of the threads stalling for the second ones. In this way the different branch paths are not executed in parallel but sequentially, and their execution time is added up.

Moreover, if one or few threads take a significantly longer route compared to any other thread during the same kernel execution, it or they will necessarily be waited by every other thread, that meanwhile is in a stall situation.

In Figure 7.5 we can see represented the reasons why threads of each kernel launch end up in a stall situation. Three main reasons can be easily identified: synchronisation, memory dependency and execution dependency.

Starting from the main one, with 41% of stalls, it is due to synchronisation issues. They happen when a thread warps are blocked on a `__syncthreads` instruction. In our code, at the end of each thread function, it is necessary to wait that every thread has finished their work, in order to retrieve to the CPU the correct best solution found during the kernel execution. But as we said many

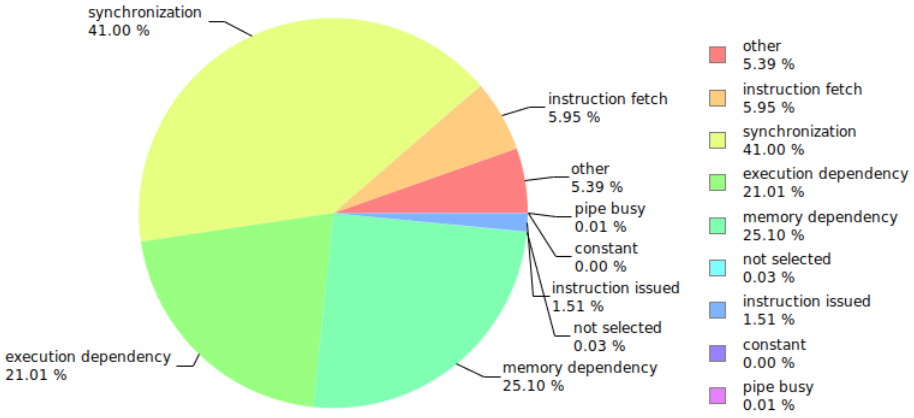


Figure 7.5: A chart representing main stall reasons of the CUDA kernel function.

times, discussing about the maximum common subgraph problem, the McSplit algorithm and its parallel implementations, the workload for each thread is inherently heavily unbalanced. Threads can compute branches of the research tree with very different depth one from each other, ending up with a situation in which the majority of threads stalls at the end of the thread function waiting that slower threads finish their work.

The second main reason of stall is due to memory dependencies, i.e. when a load or store operation needs to wait for free memory bandwidth. This situation happens frequently in our code, where data locality principle is not possible, because each thread has a separate stack to simulate recursions, that is allocated in the local memory of each thread. Thus, every time a location of the stack is accessed, a memory fetch has to be performed, and such requests cannot be coalesced within threads in the same warp, because each threads access to different cells of respective stack that are not close in memory.

The third significant reason why threads stall is due to execution dependency, i.e. when an instruction is stalled waiting for one or more arguments to be ready. It could be avoided by increasing instruction-level parallelism, but of course it is not so simple, because we also have to take into account how many registers are reserved for each thread, and the maximum number of registers available for each

Line / File	main.cu - /home/gab/cuda-workspace/Copy of v4.1.1/src
253	Divergence = 9.4% [17078 divergent executions out of 180781 total executions]
325	Divergence = 40.7% [26932 divergent executions out of 66120 total executions]
329	Divergence = 22.1% [6806 divergent executions out of 30819 total executions]
334	Divergence = 22.3% [6884 divergent executions out of 30819 total executions]

Figure 7.6: Extract from the profiling analysis executed by Nsight showing principal divergent instruction in the code.

block, that is an important limit for increasing threads number and instruction parallelism.

In addition to the stalls chart, it is also interesting to observe that some branch instructions in the kernel function show an high amount of divergence during the execution. In Figure 7.6, are indicated the most divergent instruction inside the body of the kernel function. Note that the worst one, at line 325 of the file (cf. Appendix B), correspond to the branch where we calculate the bound of the current branch and eventually prune it. Taking a route rather than the other, is significantly different, because in the first case, we execute the body of the `while` loop of the kernel function and compute new solutions and new domains as it is supposed to work the algorithm, in the other case, we prune the branch and skip to the next `while` loop iteration. The other relevant branches that present high divergence (around 20%), are again branches inside the main `while` loop of the kernel function. This lead to overall high percentage of branch divergence in the core of the kernel function, limiting a lot the maximum instruction throughput reachable by the kernel.

Chapter 8

Conclusions

The main purpose of this thesis was to explore the possibility to implement very complex subgraph algorithm in the CUDA environment. They are usually very hard to solve, and lots of attempts with different approaches have been done trying to solve them efficiently. The main issue we encountered is that often best algorithms are based on complex data structures and recursion, both features that doesn't suit well with GPU computing.

Aware that this would not have been an easy job, we decided to continue to analyse one of the most efficient existing approach, i.e. the McSplit algorithm, in order to adapt it to run in CUDA. Our goal was, to reach some interesting speedup in the CUDA implementation of the algorithm, since when it is possible and it is well applied, GPU computing can brings huge benefits on algorithms execution time.

We tried several different approaches to do so, we tried to re-implement the the McSplit algorithm using new data structures, more suitable for CUDA environment, and we explored different multi-threading paradigms, exploiting priority queues, stacks, circular buffers and other tasks management systems trying to find the best one to efficiently handle job when hundreds or thousands of threads are involved.

Unfortunately, as shown in the previous chapter, our results have clearly not been satisfying as we hoped. We hit hard against the great complexity of solving the maximum common subgraph problem on many-core architectures.

In fact we would have needed large amount of initial data to work with on

the GPU, but initial data for our problem is nothing but a couple of graphs with at most some dozens of vertices, not properly a large data-set.

We also have needed a simpler execution flow, that included only some arithmetical instruction on input data, and not a highly recursive algorithm with complex data structure to be maintained.

But, of course, if we had have all above features naturally provided by the problem, it would have been an *embarrassingly parallel problem*, and it would not have been hard to be implemented in the CUDA environment and actually not so interesting or innovative trying to do it.

Future work

Our work stops on this thesis and probably we will not continue to work on this argument, in our opinion, not so much can be done further in order to improve our code.

One of the first things that should done before trying to further develop our implementation, that we did not for lack of time, would be to perform more tests on the CUDA implementation, compared with the fastest CPU parallel one, with larger graphs and longer timeout, at least a hundred or a thousand seconds, that allows to add one or two quadrants on the right to the performances plots that we illustrated in the previous chapter. Then it will be possible to better analyse the behaviour of the implementations on hard instances, to see if somehow CUDA implementation could become advantageous compared with the CPU multi-thread version.

Given the main functioning of GPU computing, that uses SIMD paradigm, it is very difficult to adapt a inherently unbalanced recursive algorithm to such paradigm. In fact, as we said, one of the main issue of the CUDA implementation, and in our opinion the main reason why the code performs worse than what we expected, is the high branch divergence due to unbalanced workload for each GPU thread. This, in our opinion, is a main feature of the maximum common subgraph problem, and of the `McSplit` algorithm.

Anyway, supposing to hide the inefficiency due to branch divergence with a higher number of threads, it would be interesting trying to develop a new solution based on asynchronous kernel functions and a multi-thread base program

on the CPU. It would be possible to quickly produce a huge number of branch data with CPU multi-threading, let us say, for example, at the eighth depth level of the research tree. Then every time a large enough set of branch data is ready, it could be copied on the GPU and the CPU immediately start to compute other branches and spawn more kernels. In this way, more branches would be created, copied on the GPU and computed, even if they would be pruned if the correct best solution was been retrieved from the previous kernel, actually creating more useless work for the GPU. But at the same time, we could keep both the CPU and the GPU continuously busy computing branches of the research tree, updating the best solution in an asynchronous fashion.

Appendices

Appendix A

V3: sequential iterative code

```
#include <argp.h>
#include <limits.h>
#include <locale.h>
#include <stdbool.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <time.h>

#include "graph.h"

#define L   0
#define R   1
#define LL  2
#define RL  3
#define ADJ 4
#define P   5
#define W   6
#define IRL 7

#define BDS 8

#define MIN(a, b) (a < b)? a : b

#define STACK_RESIZE 4

typedef struct stack {
  unsigned pos, size;
  int (*stack)[BDS];
}stack;

static struct argp_option options[] = {
```

```
33      {"quiet", 'q', 0, 0, "Quiet output"},
34      {"verbose", 'v', 0, 0, "Verbose output"},
35      {"lad", 'l', 0, 0, "Read LAD format"},
36      {"timeout", 't', "timeout", 0, "Set timeout of TIMEOUT
                    milliseconds"},
37      {"connected", 'c', 0, 0, "Solve max common CONNECTED
                    subgraph problem"},
38      { 0 }
39  };
40
41  static char doc[] = "Find a maximum isomorphic graph";
42  static char args_doc[] = "FILENAME1 FILENAME2";
43  static struct {
44    bool quiet;
45    bool verbose;
46    bool connected;
47    bool lad;
48      int timeout;
49    char *filename1;
50    char *filename2;
51    int arg_num;
52  } arguments;
53
54  void set_default_arguments() {
55    arguments.quiet = false;
56    arguments.verbose = false;
57    arguments.lad = false;
58      arguments.timeout = 0;
59    arguments.connected = false;
60    arguments.filename1 = NULL;
61    arguments.filename2 = NULL;
62    arguments.arg_num = 0;
63  }
64  static error_t parse_opt (int key, char *arg, struct argp_state
                      *state) {
65    switch (key) {
66    case 'l':
67    arguments.lad = true;
68    break;
69    case 'q':
70      arguments.quiet = true;
71      break;
72    case 't':
73        arguments.timeout = strtol(arg, NULL, 10);
74          break;
75    case 'v':
76      arguments.verbose = true;
77      break;
78    case 'c':
```

```
79      arguments.connected = true;
80      break;
81    case ARGP_KEY_ARG:
82      if (arguments.arg_num == 0) {
83        arguments.filename1 = arg;
84      } else if (arguments.arg_num == 1) {
85        arguments.filename2 = arg;
86      } else {
87        argp_usage(state);
88      }
89      arguments.arg_num++;
90      break;
91    case ARGP_KEY_END:
92      if (arguments.arg_num == 0)
93        argp_usage(state);
94      break;
95    default: return ARGP_ERR_UNKNOWN;
96    }
97    return 0;
98  }
99  static struct argp argp = { options, parse_opt, args_doc, doc };
100
101 uchar **adjmat0, **adjmat1, n0, n1;
102 uint max_dom = 0;
103 struct timespec start;
104
105 void uchar_swap(uchar *a, uchar *b){
106   uchar tmp = *a;
107   *a = *b;
108   *b = tmp;
109 }
110
111 bool check_sol(graph_t *g0, graph_t *g1 , uchar sol[][2], uint
                       sol_len) {
112   bool *used_left = (bool*)calloc(g0->n, sizeof *used_left);
113   bool *used_right = (bool*)calloc(g1->n, sizeof *used_right);
114   for (int i = 0; i < sol_len; i++) {
115     if (used_left[sol[i][L]])  {
116       printf("node␣%d␣of␣g0␣used␣twice\n",
                       used_left[sol[i][L]]);
117       return false;
118     }
119     if (used_right[sol[i][R]])  {
120       printf("node␣%d␣of␣g1␣used␣twice\n",
                       used_right[sol[i][L]]);
121       return false;
122     }
123     used_left[sol[i][L]] = true;
124     used_right[sol[i][R]] = true;
```

```
125     if (g0->label[sol[i][L]] != g1->label[sol[i][R]]){
126       printf("g0:%d and g1:%d have different labels\n",
                        sol[i][L], sol[i][R]);
127       return false;
128     }
129     for (int j = i + 1; j < sol_len; j++) {
130       if (g0->adjmat[sol[i][L]][sol[j][L]] !=
                        g1->adjmat[sol[i][R]][sol[j][R]])
131       {
132         printf("g0(%d-%d) is different than
                        g1(%d-%d)\n",sol[i][L],sol[j][L],
                        sol[i][R], sol[j][R]);
133         return false;
134       }
135     }
136   }
137   return true;
138 }
139
140 void update_incumbent(uchar cur[][2], uchar inc[][2], uint
                        cur_pos, uint *inc_pos){
141   if(cur_pos > *inc_pos){
142     *inc_pos = cur_pos;
143     if(arguments.verbose) printf("New incumbent size: %d\n",
                        *inc_pos);
144
145     for(int i = 0; i < cur_pos; i++){
146       inc[i][L] = cur[i][L];
147       inc[i][R] = cur[i][R];
148     }
149   }
150 }
151
152 // BIDOMAINS FUNCTIONS
                        ////////////////////////////////////////////////////////////
153 void add_bidomain(uchar domains[][BDS], uint *bd_pos, uchar
                        left_i, uchar right_i, uchar left_len,
                        uchar right_len, uchar is_adjacent, uchar
                        cur_pos){
154   domains[*bd_pos][L] = left_i;
155   domains[*bd_pos][R] = right_i;
156   domains[*bd_pos][LL] = left_len;
157   domains[*bd_pos][RL] = right_len;
158   domains[*bd_pos][ADJ] = is_adjacent;
159   domains[*bd_pos][P] = cur_pos;
160   domains[*bd_pos][W] = UCHAR_MAX;
161   domains[*bd_pos][IRL] = right_len;
162   (*bd_pos)++;
163   if(*bd_pos > max_dom) max_dom = *bd_pos;
```

```
164 }
165
166 uint calc_bound(uchar domains[][BDS], uint bd_pos, uint
                    cur_pos){
167   uint bound = 0;
168   for(int i = bd_pos -1; i >= 0 && domains[i][P] == cur_pos;
                    i--)
169     bound += MIN(domains[i][LL], domains[i][IRL]);
170   return bound;
171 }
172
173 uchar partition(uchar *arr, uchar start, uchar len, const uchar
                    *adjrow){
174   uchar i = 0;
175   for(uchar j = 0; j < len; j++){
176     if(adjrow[arr[start+j]]){
177       uchar_swap(&arr[start + i], &arr[start + j]);
178       i++;
179     }
180   }
181   return i;
182 }
183
184 void generate_next_domains(uchar domains[][BDS], uint *bd_pos,
                    uint cur_pos, uchar *left, uchar *right,
                    uchar v, uchar w, uint inc_pos){
185   int i;
186   uint bd_backup = *bd_pos;
187   uint bound = 0;
188   uchar *bd;
189   for(i = *bd_pos-1, bd = &domains[i][L]; i >= 0 && bd[P] ==
                    cur_pos-1; i--, bd = &domains[i][L]){
190
191     uchar l_len = partition(left, bd[L], bd[LL], adjmat0[v]);
192     uchar r_len = partition(right, bd[R], bd[RL], adjmat1[w]);
193
194     if(bd[LL] - l_len && bd[RL] - r_len){
195       add_bidomain(domains, bd_pos, bd[L] + l_len, bd[R] +
                    r_len, bd[LL] - l_len, bd[RL]  - r_len,
                    bd[ADJ], (uchar)(cur_pos));
196       bound += MIN(bd[LL] - l_len, bd[RL]  - r_len);
197     }
198     if(l_len && r_len){
199       add_bidomain(domains, bd_pos, bd[L], bd[R], l_len, r_len,
                    true, (uchar)(cur_pos));
200       bound += MIN(l_len, r_len);
201     }
202   }
203   if (cur_pos + bound <= inc_pos)  *bd_pos = bd_backup;
```

```
204 }
205
206 uchar select_next_v(uchar *left, uchar *bd){
207   uchar min = UCHAR_MAX, idx = UCHAR_MAX;
208   if(bd[RL] != bd[IRL])
209     return left[bd[L] + bd[LL]];
210   for (uchar i = 0; i < bd[LL]; i++)
211     if (left[bd[L] + i] < min) {
212       min = left[bd[L] + i];
213       idx = i;
214     }
215   uchar_swap(&left[bd[L] + idx], &left[bd[L] + bd[LL] - 1]);
216   bd[LL]--;
217   bd[RL]--;
218   return min;
219 }
220 uchar find_min_value(uchar *arr, uchar start_idx, uchar len){
221   uchar min_v = UCHAR_MAX;
222     for(int i = 0; i < len; i++){
223         if(arr[start_idx+i] < min_v)
224             min_v = arr[start_idx + i];
225     }
226     return min_v;
227 }
228
229 void select_bidomain(uchar domains[][BDS], uint bd_pos,  uchar
                        *left, int current_matching_size, bool
                        connected){
230   int i;
231   uint min_size = UINT_MAX;
232   uint min_tie_breaker = UINT_MAX;
233   uint best = UINT_MAX;
234   uchar *bd;
235   for (i = bd_pos - 1, bd = &domains[i][L]; i >= 0 && bd[P] ==
                        current_matching_size; i--, bd =
                        &domains[i][L]) {
236     if (connected && current_matching_size>0 && !bd[ADJ])
                        continue;
237     int len = bd[LL] > bd[RL] ? bd[LL] : bd[RL];
238     if (len < min_size) {
239       min_size = len;
240       min_tie_breaker = find_min_value(left, bd[L], bd[LL]);
241       best = i;
242     } else if (len == min_size) {
243       int tie_breaker = find_min_value(left, bd[L], bd[LL]);
244       if (tie_breaker < min_tie_breaker) {
245         min_tie_breaker = tie_breaker;
246         best = i;
247       }
```

```
248       }
249     }
250     if(best != UINT_MAX && best != bd_pos-1){
251       uchar tmp[BDS];
252       for(i = 0; i < BDS; i++) tmp[i] = domains[best][i];
253       for(i = 0; i < BDS; i++) domains[best][i] =
                            domains[bd_pos-1][i];
254       for(i = 0; i < BDS; i++) domains[bd_pos-1][i] = tmp[i];
255
256     }
257   }
258
259   double compute_elapsed_sec(){
260     struct timespec now;
261     double time_elapsed;
262
263     clock_gettime(CLOCK_MONOTONIC, &now);
264     time_elapsed = (now.tv_sec - start.tv_sec);
265     time_elapsed += (double)(now.tv_nsec - start.tv_nsec) /
                            1000000000.0;
266
267     return time_elapsed;
268   }
269
270   uchar select_next_w(uchar *right, uchar *bd) {
271     uchar min = UCHAR_MAX, idx = UCHAR_MAX;
272     for (uchar i = 0; i < bd[RL]+1; i++)
273       if ((right[bd[R] + i] > bd[W] || bd[W] == UCHAR_MAX)
274           && right[bd[R] + i] < min) {
275         min = right[bd[R] + i];
276         idx = i;
277       }
278     if(idx == UCHAR_MAX)
279       bd[RL]++;
280     return idx;
281   }
282
283   void mcs(uchar incumbent[][2], uint *inc_pos){
284
285     uint min = MIN(n0, n1);
286
287     uchar cur[min][2];
288     uchar domains[min*min][BDS];
289     uchar left[n0], right[n1];
290     uchar v, w, *bd;
291     uint bd_pos = 0;
292     for(uchar i = 0; i < n0; i++) left[i] = i;
293     for(uchar i = 0; i < n1; i++) right[i] = i;
294     add_bidomain(domains, &bd_pos, 0, 0, n0, n1, 0, 0);
```

```
295
296   while (bd_pos > 0) {
297     if (arguments.timeout && compute_elapsed_sec() >
                      arguments.timeout) {
298           arguments.timeout = -1;
299           return;
300       }
301
302     bd = &domains[bd_pos - 1][L];
303     if (calc_bound(domains, bd_pos, bd[P]) + bd[P] <= *inc_pos
                      || (bd[LL] == 0 && bd[RL] == bd[IRL])) {
304       bd_pos--;
305     } else {
306       select_bidomain(domains, bd_pos, left, domains[bd_pos -
                      1][P], arguments.connected);
307       v = select_next_v(left, bd);
308       if ((bd[W] = select_next_w(right, bd)) != UCHAR_MAX) {
309         w = right[bd[R] + bd[W]];       // swap the W after the
                      bottom of the current right domain
310         right[bd[R] + bd[W]] = right[bd[R] + bd[RL]];
311         right[bd[R] + bd[RL]] = w;
312         bd[W] = w;                      // store the W used for
                      this iteration
313         cur[bd[P]][L] = v;
314         cur[bd[P]][R] = w;
315         update_incumbent(cur, incumbent, bd[P] + (uchar) 1,
                      inc_pos);
316         generate_next_domains(domains, &bd_pos, bd[P] + 1,
                      left, right, v, w, *inc_pos);
317       }
318     }
319   }
320 }
321
322 int main(int argc, char** argv){
323   set_default_arguments();
324   argp_parse(&argp, argc, argv, 0, 0, 0);
325   struct timespec finish;
326   double time_elapsed;
327
328
329   char format = arguments.lad ? 'L' : 'B';
330   graph_t *g0 = calloc(1, sizeof *g0 );
331   readGraph(arguments.filename1, g0, format);
332   graph_t *g1 = calloc(1, sizeof *g1 );
333   readGraph(arguments.filename2, g1, format);
334   g0 = sort_vertices_by_degree(g0, (graph_edge_count(g1) >
                      g1->n*(g1->n-1)/2));
335   g1 = sort_vertices_by_degree(g1, (graph_edge_count(g0) >
```

```
                        g0->n*(g0->n-1)/2));
336
337     adjmat0 = g0->adjmat;
338     adjmat1 = g1->adjmat;
339
340     n0 = g0->n;
341     n1 = g1->n;
342     uint min_size = MIN(n0, n1);
343     uchar solution[min_size][2];
344
345     uint sol_len = 0;
346     clock_gettime(CLOCK_MONOTONIC, &start);
347     mcs(solution, &sol_len);
348     clock_gettime(CLOCK_MONOTONIC, &finish);
349
350
351
352
353     if (!check_sol(g0, g1, solution, sol_len)) {
354       fprintf(stderr, "***␣Error:␣Invalid␣solution\n");
355     }
356
357     if (arguments.timeout == -1){
358         printf("TIMEOUT\n");
359     }
360
361     printf("SOLUTION␣size:%d\nsol:␣", sol_len);
362     for(int i = 0; i < g0->n; i++)
363       for(int j = 0; j < sol_len; j++)
364         if(solution[j][L] == i)
365           printf("|%2d␣%2d|␣", solution[j][L], solution[j][R]);
366     printf("\n");
367
368     time_elapsed = (finish.tv_sec - start.tv_sec); // calculating
                        elapsed seconds
369     time_elapsed += (double)(finish.tv_nsec - start.tv_nsec) /
                        1000000000.0; // adding elapsed nanoseconds
370     printf(">>>␣%d␣-␣%015.10f", sol_len, time_elapsed);
371
372     free_graph(g0);
373     free_graph(g1);
374     return 0;
375 }
```

Appendix B

V5: CUDA implementation

```
#include <argp.h>
#include <limits.h>
#include <locale.h>
#include <stdbool.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <time.h>

#include "graph.h"

#define L   0
#define R   1
#define LL  2
#define RL  3
#define ADJ 4
#define P   5
#define W   6
#define IRL 7

#define BDS 8

#define START 0
#define END 1

#define MIN(a, b) (a < b)? a : b

#define N_BLOCKS 64
#define BLOCK_SIZE 512
#define MAX_GRAPH_SIZE 64
#define checkCudaErrors(value)
                    CheckCudaErrorAux(__FILE__,__LINE__,
```

```
                        #value, value)
32
33  typedef unsigned char uchar;
34  typedef unsigned int uint;
35
36  __constant__ uchar d_adjmat0[MAX_GRAPH_SIZE][MAX_GRAPH_SIZE];
37  __constant__ uchar d_adjmat1[MAX_GRAPH_SIZE][MAX_GRAPH_SIZE];
38  __constant__ uchar d_n0;
39  __constant__ uchar d_n1;
40
41  uchar adjmat0[MAX_GRAPH_SIZE][MAX_GRAPH_SIZE];
42  uchar adjmat1[MAX_GRAPH_SIZE][MAX_GRAPH_SIZE];
43  uchar n0;
44  uchar n1;
45
46  uint __gpu_level = 5;
47  struct timespec start;
48
49  static struct argp_option options[] = {
50      { "verbose", 'v', 0, 0, "Verbose output" },
51      { "lad", 'l', 0, 0, "Read LAD format"},
52      { "timeout", 't', "timeout", 0, "Set timeout of TIMEOUT
                        milliseconds"},
53      { "connected", 'c', 0, 0, "Solve max common CONNECTED
                        subgraph problem" },
54      { 0 }
55  };
56
57  static char doc[] = "Find a maximum isomorphic graph";
58  static char args_doc[] = "FILENAME1 FILENAME2";
59  static struct {
60    bool verbose;
61    bool lad;
62    bool connected;
63    int timeout;
64    char *filename1;
65    char *filename2;
66    int arg_num;
67  } arguments;
68  void set_default_arguments() {
69    arguments.verbose = false;
70    arguments.lad = false;
71    arguments.timeout = 0;
72    arguments.connected = false;
73    arguments.filename1 = NULL;
74    arguments.filename2 = NULL;
75    arguments.arg_num = 0;
76  }
77  static error_t parse_opt(int key, char *arg, struct argp_state
```

```
                        *state) {
78    switch (key) {
79    case 'v':
80      arguments.verbose = true;
81      break;
82    case 't':
83      arguments.timeout = strtol(arg, NULL, 10);
84      break;
85    case 'l':
86      arguments.lad = true;
87      break;
88    case 'c':
89      arguments.connected = true;
90      break;
91    case ARGP_KEY_ARG:
92      if (arguments.arg_num == 0) {
93        arguments.filename1 = arg;
94      } else if (arguments.arg_num == 1) {
95        arguments.filename2 = arg;
96      } else {
97        argp_usage(state);
98      }
99      arguments.arg_num++;
100     break;
101   case ARGP_KEY_END:
102     if (arguments.arg_num == 0)
103       argp_usage(state);
104     break;
105   default:
106     return ARGP_ERR_UNKNOWN;
107   }
108   return 0;
109 }
110
111 __host__ __device__
112 void uchar_swap(uchar *a, uchar *b){
113   uchar tmp = *a;
114   *a = *b;
115   *b = tmp;
116 }
117 __host__ __device__
118 uchar select_next_v(uchar *left, uchar *bd){
119   uchar min = UCHAR_MAX, idx = UCHAR_MAX;
120   if(bd[RL] != bd[IRL])
121     return left[bd[L] + bd[LL]];
122   for (uchar i = 0; i < bd[LL]; i++)
123     if (left[bd[L] + i] < min) {
124       min = left[bd[L] + i];
125       idx = i;
```

```
126       }
127     uchar_swap(&left[bd[L] + idx], &left[bd[L] + bd[LL] - 1]);
128     bd[LL]--;
129     bd[RL]--;
130     return min;
131   }
132
133
134   __host__ __device__
135   uchar select_next_w(uchar *right, uchar *bd) {
136     uchar min = UCHAR_MAX, idx = UCHAR_MAX;
137     for (uchar i = 0; i < bd[RL]+1; i++)
138       if ((right[bd[R] + i] > bd[W] || bd[W] == UCHAR_MAX)
139           && right[bd[R] + i] < min) {
140         min = right[bd[R] + i];
141         idx = i;
142       }
143     if(idx == UCHAR_MAX)
144       bd[RL]++;
145     return idx;
146   }
147
148   __host__  __device__ uchar index_of_next_smallest(const uchar
                          *arr,
149       uchar start_idx, uchar len, uchar w) {
150     uchar idx = UCHAR_MAX;
151     uchar smallest = UCHAR_MAX;
152     for (uchar i = 0; i < len; i++) {
153       if ((arr[start_idx + i] > w || w == UCHAR_MAX)
154           && arr[start_idx + i] < smallest) {
155         smallest = arr[start_idx + i];
156         idx = i;
157       }
158     }
159     return idx;
160   }
161
162   __host__  __device__ uchar find_min_value(const uchar *arr,
                          uchar start_idx,
163       uchar len) {
164     uchar min_v = UCHAR_MAX;
165     for (int i = 0; i < len; i++) {
166       if (arr[start_idx + i] < min_v)
167         min_v = arr[start_idx + i];
168     }
169     return min_v;
170   }
171
172   __host__ __device__
```

```
173 void remove_from_domain(uchar *arr, const uchar *start_idx,
                          uchar *len,
174     uchar v) {
175   int i = 0;
176   for (i = 0; arr[*start_idx + i] != v; i++)
177     ;
178   uchar_swap(&arr[*start_idx + i], &arr[*start_idx + *len - 1]);
179   (*len)--;
180 }
181
182 __host__ __device__
183 void update_incumbent(uchar cur[][2], uchar inc[][2], uchar
                          cur_pos,
184     uchar *inc_pos) {
185   if (cur_pos > *inc_pos) {
186     *inc_pos = cur_pos;
187     for (int i = 0; i < cur_pos; i++) {
188       inc[i][L] = cur[i][L];
189       inc[i][R] = cur[i][R];
190     }
191   }
192 }
193
194 // BIDOMAINS FUNCTIONS
                          ////////////////////////////////////////////////////////////
195 __host__ __device__
196 void add_bidomain(uchar domains[][BDS], uint *bd_pos, uchar
                          left_i,
197     uchar right_i, uchar left_len, uchar right_len, uchar
                          is_adjacent,
198     uchar cur_pos) {
199   domains[*bd_pos][L]   = left_i;
200   domains[*bd_pos][R]   = right_i;
201   domains[*bd_pos][LL]  = left_len;
202   domains[*bd_pos][RL]  = right_len;
203   domains[*bd_pos][ADJ]   = is_adjacent;
204   domains[*bd_pos][P]   = cur_pos;
205   domains[*bd_pos][W]   = UCHAR_MAX;
206   domains[*bd_pos][IRL]   = right_len;
207
208   (*bd_pos)++;
209 }
210
211 __host__  __device__ uint calc_bound(uchar domains[][BDS], uint
                          bd_pos,
212     uint cur_pos, uint *bd_n) {
213   uint bound = 0;
214   int i;
215   for (i = bd_pos - 1; i >= 0 && domains[i][P] == cur_pos; i--)
```

```
216     bound += MIN(domains[i][LL], domains[i][IRL]);
217   *bd_n = bd_pos - 1 - i;
218   return bound;
219 }
220
221 __host__  __device__ uchar partition(uchar *arr, uchar start,
                       uchar len,
222     const uchar *adjrow) {
223   uchar i = 0;
224   for (uchar j = 0; j < len; j++) {
225     if (adjrow[arr[start + j]]) {
226       uchar_swap(&arr[start + i], &arr[start + j]);
227       i++;
228     }
229   }
230   return i;
231 }
232
233 __host__  __device__
234 uchar find_min_value(uchar *arr, uchar start_idx, uchar len){
235   uchar min_v = UCHAR_MAX;
236   for(int i = 0; i < len; i++){
237     if(arr[start_idx+i] < min_v)
238       min_v = arr[start_idx + i];
239   }
240   return min_v;
241 }
242
243 __host__  __device__
244 void select_bidomain(uchar domains[][BDS], uint bd_pos,  uchar
                       *left, int current_matching_size, bool
                       connected){
245   int i;
246   uint min_size = UINT_MAX;
247   uint min_tie_breaker = UINT_MAX;
248   uint best = UINT_MAX;
249   uchar *bd;
250   for (i = bd_pos - 1, bd = &domains[i][L]; i >= 0 && bd[P] ==
                       current_matching_size; i--, bd =
                       &domains[i][L]) {
251     if (connected && current_matching_size>0 && !bd[ADJ])
                       continue;
252     int len = bd[LL] > bd[RL] ? bd[LL] : bd[RL];
253     if (len < min_size) {
254       min_size = len;
255       min_tie_breaker = find_min_value(left, bd[L], bd[LL]);
256       best = i;
257     } else if (len == min_size) {
258       int tie_breaker = find_min_value(left, bd[L], bd[LL]);
```

```
259       if (tie_breaker < min_tie_breaker) {
260         min_tie_breaker = tie_breaker;
261         best = i;
262       }
263     }
264   }
265   if(best != UINT_MAX && best != bd_pos-1){
266     uchar tmp[BDS];
267     for(i = 0; i < BDS; i++) tmp[i] = domains[best][i];
268     for(i = 0; i < BDS; i++) domains[best][i] =
                        domains[bd_pos-1][i];
269     for(i = 0; i < BDS; i++) domains[bd_pos-1][i] = tmp[i];
270
271   }
272 }
273
274
275
276 __device__
277 void d_generate_next_domains(uchar domains[][BDS], uint
                        *bd_pos, uint cur_pos, uchar *left, uchar
                        *right, uchar v, uchar w, uint inc_pos) {
278   int i;
279   uint bd_backup = *bd_pos;
280   uint bound = 0;
281   uchar *bd;
282   for (i = *bd_pos - 1, bd = &domains[i][L]; i >= 0 && bd[P] ==
                        cur_pos - 1; i--, bd = &domains[i][L]) {
283
284     uchar l_len = partition(left, bd[L], bd[LL], d_adjmat0[v]);
285     uchar r_len = partition(right, bd[R], bd[RL], d_adjmat1[w]);
286
287     if (bd[LL] - l_len && bd[RL] - r_len) {
288       add_bidomain(domains, bd_pos, bd[L] + l_len, bd[R] +
                        r_len, bd[LL] - l_len, bd[RL] - r_len,
                        bd[ADJ], (uchar) (cur_pos));
289       bound += MIN(bd[LL] - l_len, bd[RL] - r_len);
290     }
291     if (l_len && r_len) {
292       add_bidomain(domains, bd_pos, bd[L], bd[R], l_len, r_len,
                        true, (uchar) (cur_pos));
293       bound += MIN(l_len, r_len);
294     }
295   }
296   if (cur_pos + bound <= inc_pos)
297     *bd_pos = bd_backup;
298 }
299
300 __global__
```

```
301 void d_mcs(uchar *args, uint n_threads, uchar a_size, uint
                    *args_i, uint actual_inc, uchar
                    *device_solutions, uint max_sol_size, uint
                    last_arg, bool verbose, bool connected) {
302   uint my_idx = (blockIdx.x * blockDim.x) + threadIdx.x;
303   uchar cur[MAX_GRAPH_SIZE][2], incumbent[MAX_GRAPH_SIZE][2],
304   domains[MAX_GRAPH_SIZE * 5][BDS], left[MAX_GRAPH_SIZE],
305   right[MAX_GRAPH_SIZE], v, w;
306   uint bd_pos = 0, bd_n = 0;
307   uchar inc_pos = 0;
308   __shared__ uint sh_inc;
309   sh_inc = actual_inc;
310   __syncthreads();
311   if (my_idx < n_threads) {
312     for (int i = args_i[my_idx]; i < last_arg && ( my_idx <
                    n_threads-1 &&  i < args_i[my_idx +1]);) {
313       add_bidomain(domains, &bd_pos, args[i++], args[i++],
                    args[i++], args[i++], args[i++],
                    args[i++]);
314       for (int p = 0; p < domains[bd_pos - 1][P]; p++)
315         cur[p][L] = args[i++];
316       for (int p = 0; p < domains[bd_pos - 1][P]; p++)
317         cur[p][R] = args[i++];
318       for (int l = 0; l < d_n0; l++)
319         left[l] = args[i++];
320       for (int r = 0; r < d_n1; r++)
321         right[r] = args[i++];
322     }
323     while (bd_pos > 0) {
324       uchar *bd = &domains[bd_pos - 1][L];
325       if (calc_bound(domains, bd_pos, bd[P], &bd_n) + bd[P] <=
                    sh_inc || (bd[LL] == 0 && bd[RL] ==
                    bd[IRL])) {
326         bd_pos--;
327       } else {
328         select_bidomain(domains, bd_pos, left, domains[bd_pos -
                    1][P], connected);
329         if (bd[RL] == bd[IRL]) {
330           v = find_min_value(left, bd[L], bd[LL]);
331           remove_from_domain(left, &bd[L], &bd[LL], v);
332           bd[RL]--;
333         } else v = left[bd[L] + bd[LL]];
334         if ((bd[W] = index_of_next_smallest(right, bd[R],
                    bd[RL] + (uchar) 1, bd[W])) == UCHAR_MAX) {
335           bd[RL]++;
336         } else {
337           w = right[bd[R] + bd[W]];
338           right[bd[R] + bd[W]] = right[bd[R] + bd[RL]];
339           right[bd[R] + bd[RL]] = w;
```

```
340             bd[W] = w;
341             cur[bd[P]][L] = v;
342             cur[bd[P]][R] = w;
343             update_incumbent(cur, incumbent, bd[P] + 1, &inc_pos);
344             atomicMax(&sh_inc, inc_pos);
345             d_generate_next_domains(domains, &bd_pos, bd[P] + 1,
                            left, right, v, w, inc_pos);
346           }
347         }
348       }
349     }
350     device_solutions[blockIdx.x* max_sol_size] = 0;
351
352     __syncthreads();
353     if (atomicCAS(&sh_inc, inc_pos, 0) == inc_pos && inc_pos > 0)
                            {
354       if(verbose) printf("Th␣%d␣found␣new␣solution␣of␣size␣%d\n",
                            my_idx, inc_pos);
355       bd_pos = 0;
356       device_solutions[blockIdx.x* max_sol_size + bd_pos++] =
                            inc_pos;
357       for (int i = 0; i < inc_pos; i++)
358         device_solutions[blockIdx.x* max_sol_size + bd_pos++] =
                            incumbent[i][L];
359       for (int i = 0; i < inc_pos; i++)
360         device_solutions[blockIdx.x* max_sol_size + bd_pos++] =
                            incumbent[i][R];
361     }
362   }
363
364   double compute_elapsed_sec(struct timespec strt){
365     struct timespec now;
366     double time_elapsed;
367
368     clock_gettime(CLOCK_MONOTONIC, &now);
369     time_elapsed = (now.tv_sec - strt.tv_sec);
370     time_elapsed += (double)(now.tv_nsec - strt.tv_nsec) /
                            1000000000.0;
371
372     return time_elapsed;
373   }
374
375   static void CheckCudaErrorAux(const char *file, unsigned line,
376       const char *statement, cudaError_t err) {
377     if (err == cudaSuccess)
378       return;
379     fprintf(stderr, "%s␣returned␣%s(%d)␣at␣%s:%d\n", statement,
380         cudaGetErrorString(err), err, file, line);
381     exit(1);
```

```
382 }
383
384 void move_graphs_to_gpu(graph_t *g0, graph_t *g1) {
385   checkCudaErrors(cudaMemcpyToSymbol(d_n0, &g0->n,
                      sizeof(uchar)));
386   checkCudaErrors(cudaMemcpyToSymbol(d_n1, &g1->n,
                      sizeof(uchar)));
387
388   checkCudaErrors(cudaMemcpyToSymbol(d_adjmat0, adjmat0,
                      MAX_GRAPH_SIZE*MAX_GRAPH_SIZE));
389   checkCudaErrors(cudaMemcpyToSymbol(d_adjmat1, adjmat1,
                      MAX_GRAPH_SIZE*MAX_GRAPH_SIZE));
390 }
391
392
393 void h_generate_next_domains(uchar domains[][BDS], uint
                      *bd_pos, uint cur_pos,
394     uchar *left, uchar *right, uchar v, uchar w, uint inc_pos) {
395   int i;
396   uint bd_backup = *bd_pos;
397   uint bound = 0;
398   uchar *bd;
399   for (i = *bd_pos - 1, bd = &domains[i][L]; i >= 0 && bd[P] ==
                      cur_pos - 1;
400       i--, bd = &domains[i][L]) {
401
402     uchar l_len = partition(left, bd[L], bd[LL], adjmat0[v]);
403     uchar r_len = partition(right, bd[R], bd[RL], adjmat1[w]);
404
405     if (bd[LL] - l_len && bd[RL] - r_len) {
406       add_bidomain(domains, bd_pos, bd[L] + l_len, bd[R] +
                      r_len,
407           bd[LL] - l_len, bd[RL] - r_len, bd[ADJ], (uchar)
                      (cur_pos));
408       bound += MIN(bd[LL] - l_len, bd[RL] - r_len);
409     }
410     if (l_len && r_len) {
411       add_bidomain(domains, bd_pos, bd[L], bd[R], l_len, r_len,
                      true,
412           (uchar) (cur_pos));
413       bound += MIN(l_len, r_len);
414     }
415   }
416   if (cur_pos + bound <= inc_pos)
417     *bd_pos = bd_backup;
418 }
419
420
421 bool check_sol(graph_t *g0, graph_t *g1, uchar sol[][2], uint
```

```
                      sol_len) {
422   bool *used_left = (bool*) calloc(g0->n, sizeof *used_left);
423   bool *used_right = (bool*) calloc(g1->n, sizeof *used_right);
424   for (int i = 0; i < sol_len; i++) {
425     if (used_left[sol[i][L]]) {
426       printf("node␣%d␣of␣g0␣used␣twice\n",
                      used_left[sol[i][L]]);
427       return false;
428     }
429     if (used_right[sol[i][R]]) {
430       printf("node␣%d␣of␣g1␣used␣twice\n",
                      used_right[sol[i][L]]);
431       return false;
432     }
433     used_left[sol[i][L]] = true;
434     used_right[sol[i][R]] = true;
435     if (g0->label[sol[i][L]] != g1->label[sol[i][R]]) {
436       printf("g0:%d␣and␣g1:%d␣have␣different␣labels\n",
                      sol[i][L],
437           sol[i][R]);
438       return false;
439     }
440     for (int j = i + 1; j < sol_len; j++) {
441       if (g0->adjmat[sol[i][L]][sol[j][L]]
442                                  !=
                      g1->adjmat[sol[i][R]][sol[j][R]]) {
443         printf("g0(%d-%d)␣is␣different␣than␣g1(%d-%d)\n",
                      sol[i][L],
444             sol[j][L], sol[i][R], sol[j][R]);
445         return false;
446       }
447     }
448   }
449   return true;
450 }
451
452 static struct argp argp = { options, parse_opt, args_doc, doc };
453
454 void launch_kernel(uchar *args, uint n_threads, uchar a_size,
                      uint sol_size, uint *args_i,
455     uchar incumbent[][2], uchar *inc_pos, uint total_args_size,
                      uint last_arg) {
456   uchar *device_args;
457   uchar *device_solutions;
458   uchar *host_solutions;
459   uint *device_args_i;
460   uint max_sol_size = 1 + 2 * (MIN(n0, n1));
461   struct timespec sleep;
462   sleep.tv_sec = 0;
```

```
463     sleep.tv_nsec = 2000;
464     cudaEvent_t stop;
465
466     host_solutions = (uchar*) malloc(N_BLOCKS * max_sol_size *
                        sizeof *host_solutions);
467
468     checkCudaErrors(cudaEventCreate(&stop));
469
470     checkCudaErrors(cudaMalloc(&device_args, total_args_size *
                        sizeof *device_args));
471     checkCudaErrors(cudaMalloc(&device_solutions, N_BLOCKS *
                        max_sol_size * sizeof *device_solutions));
472
473
474     checkCudaErrors(cudaMemcpy(device_args, args, total_args_size
                        * sizeof *device_args,
                        cudaMemcpyHostToDevice));
475
476     checkCudaErrors(cudaMalloc(&device_args_i, N_BLOCKS *
                        BLOCK_SIZE * sizeof *device_args_i));
477     checkCudaErrors(cudaMemcpy(device_args_i, args_i, N_BLOCKS *
                        BLOCK_SIZE * sizeof *device_args_i,
                        cudaMemcpyHostToDevice));
478
479     if(arguments.verbose) printf("Launching kernel...\n");
480
481     d_mcs<<<N_BLOCKS, BLOCK_SIZE>>>(device_args, n_threads,
                        a_size, device_args_i, *inc_pos,
                        device_solutions, max_sol_size, last_arg,
                        arguments.verbose, arguments.connected);
482
483     checkCudaErrors(cudaEventRecord(stop));
484
485     while(cudaEventQuery(stop) == cudaErrorNotReady){
486       nanosleep(&sleep, NULL);
487       if(arguments.timeout && compute_elapsed_sec(start) >
                        arguments.timeout){
488         checkCudaErrors(cudaDeviceReset());
489         return;
490       }
491     }
492     if(arguments.verbose) printf("Kernel executed...\n");
493
494     checkCudaErrors(cudaMemcpy(host_solutions, device_solutions,
                        N_BLOCKS * max_sol_size * sizeof
                        *device_solutions,
                        cudaMemcpyDeviceToHost));
495
496     checkCudaErrors(cudaFree(device_args));
```

```
497   checkCudaErrors(cudaFree(device_args_i));
498
499   for(int b = 0; b < N_BLOCKS; b++){
500     if(arguments.verbose)printf("args[%d] = %d\n",
                      b*max_sol_size,
                      host_solutions[b*max_sol_size]);
501     if (*inc_pos < host_solutions[b*max_sol_size]) {
502       *inc_pos = host_solutions[b*max_sol_size];
503       for (int i = 1; i < *inc_pos + 1; i++) {
504         incumbent[i - 1][L] = host_solutions[b*max_sol_size +
                      i];
505         incumbent[i - 1][R] = host_solutions[b*max_sol_size +
                      *inc_pos + i];
506         if(arguments.verbose) printf("|%d %d| ",
                      incumbent[i-1][L], incumbent[i-1][R]);
507       }
508     }if(arguments.verbose) printf("\n");
509   }
510   free(host_solutions);
511 }
512
513
514 void fill_args(uchar *args, uchar *args_i[], uchar *bd, uchar
                      *cur[], uchar *left, uchar *right, uint
                      *n_args, uint ){
515
516 }
517
518 void *safe_realloc(void* old, uint new_size){
519   void *tmp = realloc(old, new_size);
520   if (tmp != NULL) return tmp;
521   else exit(-1);
522 }
523
524 void mcs(uchar incumbent[][2], uchar *inc_pos) {
525   uint bd_pos = 0, bd_n = 0;
526   uchar cur[MAX_GRAPH_SIZE][2], domains[MAX_GRAPH_SIZE *
                      5][BDS], left[n0],
527   right[n1], v, w;
528   for (uchar i = 0; i < n0; i++)
529     left[i] = i;
530   for (uchar i = 0; i < n1; i++)
531     right[i] = i;
532   add_bidomain(domains, &bd_pos, 0, 0, n0, n1, 0, 0);
533   //supposing an initial average of 2 domains for thread, it
                      will be reallocated if necessary
534   uint args_num = N_BLOCKS * BLOCK_SIZE * 2;
535   uint a_size = (BDS - 2 + 2 * __gpu_level + n0 + n1);
536   uint sol_size = 1 + 2*(MIN(n0, n1));
```

```
537   uint args_size = args_num * a_size;
538
539   uint args_i[N_BLOCKS * BLOCK_SIZE];
540   uchar *args = (uchar*) malloc(args_size * sizeof *args);
541   uint n_args = 0, n_threads = 0;
542
543   while (bd_pos > 0) {
544     if (arguments.timeout && compute_elapsed_sec(start) >
                    arguments.timeout) {
545       arguments.timeout = -1;
546       return;
547     }
548     uchar *bd = &domains[bd_pos - 1][L];
549
550     if (calc_bound(domains, bd_pos, bd[P], &bd_n) + bd[P] <=
                    *inc_pos || (bd[LL] == 0 && bd[RL] ==
                    bd[IRL])) {
551       bd_pos--;
552       continue;
553     }
554
555     if (bd[P] == __gpu_level) {
556       if (n_args + bd_n > args_num) {
557         args_num  = n_args + bd_n;
558         args_size = args_num * a_size;
559         args = (uchar*) safe_realloc(args, args_size * sizeof
                    *args);
560       }
561
562       args_i[n_threads] = n_args * a_size;
563
564       for (uint b = 0; b < bd_n; b++, n_args++, bd_pos--) {
565         uint arg_i = n_args * a_size, i = 0;
566         for (i = 0; i < BDS - 2; i++, arg_i++)
567           args[arg_i] = domains[bd_pos - 1][i];
568         for (i = 0; i < __gpu_level; i++, arg_i++)
569           args[arg_i] = cur[i][L];
570         for (i = 0; i < __gpu_level; i++, arg_i++)
571           args[arg_i] = cur[i][R];
572         for (i = 0; i < n0; i++, arg_i++)
573           args[arg_i] = left[i];
574         for (i = 0; i < n0; i++, arg_i++)
575           args[arg_i] = right[i];
576       }
577       n_threads++;
578       if (n_threads == N_BLOCKS * BLOCK_SIZE) {
579         launch_kernel(args, n_threads, a_size, sol_size,
                    args_i, incumbent, inc_pos, args_size,
                    n_args*a_size);
```

```
580           n_threads = 0;
581           n_args = 0;
582         }
583         continue;
584       }
585
586       select_bidomain(domains, bd_pos, left, domains[bd_pos -
                        1][P], arguments.connected);
587       if (bd[RL] == bd[IRL]) {
588         v = find_min_value(left, bd[L], bd[LL]);
589         remove_from_domain(left, &bd[L], &bd[LL], v);
590         bd[RL]--;
591       } else v = left[bd[L] + bd[LL]];
592
593
594       if ((bd[W] = index_of_next_smallest(right, bd[R], bd[RL] +
                        (uchar) 1, bd[W])) == UCHAR_MAX) {
595         bd[RL]++;
596       } else {
597         w = right[bd[R] + bd[W]];
598         right[bd[R] + bd[W]] = right[bd[R] + bd[RL]];
599         right[bd[R] + bd[RL]] = w;
600
601         bd[W] = w;
602
603         cur[bd[P]][L] = v;
604         cur[bd[P]][R] = w;
605
606         update_incumbent(cur, incumbent, bd[P] + 1, inc_pos);
607         h_generate_next_domains(domains, &bd_pos, bd[P] + 1,
                        left, right, v,
608             w, *inc_pos);
609       }
610
611     }
612     if (n_threads > 0)
613       launch_kernel(args, n_threads, a_size, sol_size, args_i,
                        incumbent, inc_pos, args_size,
                        n_args*a_size);
614   }
615
616   int main(int argc, char** argv) {
617     set_default_arguments();
618     argp_parse(&argp, argc, argv, 0, 0, 0);
619     struct timespec finish;
620     double time_elapsed;
621     char format = arguments.lad ? 'L' : 'B';
622     graph_t *g0 = (graph_t*) calloc(1, sizeof *g0);
623     readGraph(arguments.filename1, g0, format);
```

```
624   graph_t *g1 = (graph_t*) calloc(1, sizeof *g1);
625   readGraph(arguments.filename2, g1, format);
626   g0 = sort_vertices_by_degree(g0,
627       (graph_edge_count(g1) > g1->n * (g1->n - 1) / 2));
628   g1 = sort_vertices_by_degree(g1,
629       (graph_edge_count(g0) > g0->n * (g0->n - 1) / 2));
630   int min_size = MIN(g0->n, g1->n);
631   n0 = g0->n;
632   n1 = g1->n;
633
634   for (int i = 0; i < n0; i++)
635     for (int j = 0; j < n0; j++)
636       adjmat0[i][j] = g0->adjmat[i][j];
637
638   for (int i = 0; i < n1; i++)
639     for (int j = 0; j < n1; j++)
640       adjmat1[i][j] = g1->adjmat[i][j];
641
642   checkCudaErrors(cudaDeviceReset());
643   move_graphs_to_gpu(g0, g1);
644
645   uchar solution[min_size][2];
646   uchar sol_len = 0;
647   clock_gettime(CLOCK_MONOTONIC, &start);
648   mcs(solution, &sol_len);
649   clock_gettime(CLOCK_MONOTONIC, &finish);
650
651   if(arguments.timeout == -1){
652     printf("TIMEOUT\n");
653   }
654
655   printf("SOLUTION␣size:%d\nsol:␣", sol_len);
656   //for (int i = 0; i < g0->n; i++)
657   for (int j = 0; j < sol_len; j++)
658     //if (solution[j][L] == i)
659     printf("|%2d␣%2d|␣", solution[j][L], solution[j][R]);
660   printf("\n");
661
662   if (!check_sol(g0, g1, solution, sol_len)) {
663     printf("***␣Error:␣Invalid␣solution\n");
664   }
665   time_elapsed = (finish.tv_sec - start.tv_sec); // calculating
                       elapsed seconds
666   time_elapsed += (double) (finish.tv_nsec - start.tv_nsec) /
                       1000000000.0; // adding elapsed nanoseconds
667   printf(">>>␣%d␣-␣%015.10f\n", sol_len, time_elapsed);
668
669   free_graph(g0);
670   free_graph(g1);
```

```
671     return 0;
672 }
```

Appendix C

Graph class

```
1
2  #ifndef GRAPH_H_
3  #define GRAPH_H_
4
5  #include <stdio.h>
6  #include <stdlib.h>
7  #include <limits.h>
8  #include <stdbool.h>
9
10 #define INSERTION_SORT(type, arr, arr_len, swap_condition) do {
                        \
11     for (int i=1; i<arr_len; i++) {
                        \
12       for (int j=i; j>=1; j--) {                                  \
13         if (swap_condition) {                                     \
14           type tmp = arr[j-1];                                  \
15           arr[j-1] = arr[j];                                    \
16           arr[j] = tmp;                                         \
17         } else {                                                  \
18           break;                                                \
19         }                                                         \
20       }                                                             \
21     }
                        \
22 } while(0);
23
24 typedef unsigned long long ULL;
25
26 typedef struct graph_s {
27     int n;
28     unsigned char **adjmat;
29     unsigned int *label;
```

```
30      unsigned int *degree;
31  }graph_t;
32
33  unsigned int* calculate_degrees(graph_t *g);
34
35  graph_t *induced_subgraph(graph_t *g, int *vv);
36
37  int graph_edge_count(graph_t *g);
38
39  // Precondition: *g is already zeroed out
40  void readGraph(char* filename, graph_t* g, char format);
41
42  // Precondition: *g is already zeroed out
43  void readBinaryGraph(char* filename, graph_t* g);
44
45  // Precondition: *g is already zeroed out
46  void readLadGraph(char* filename, graph_t* g);
47
48  void free_graph(graph_t *g);
49
50  graph_t *sort_vertices_by_degree(graph_t *g, bool ascending );
51  //=============================================================
52
53  #include "graph.h"
54
55  static void fail(const char* msg) {
56      fprintf(stderr,"%s\n", msg);
57      exit(1);
58  }
59
60  unsigned int* calculate_degrees(graph_t *g) {
61    short int size = g->n;
62    unsigned int *degree = calloc(size, sizeof *degree);
63    for (int v = 0; v < g->n; v++)
64      for (int w = 0; w < g->n; w++)
65        if (g->adjmat[v][w]) degree[v]++;
66    return degree;
67  }
68
69  void add_edge(graph_t *g, int v, int w) {
70      if (v != w) {
71          g->adjmat[v][w] = 1;
72          g->adjmat[w][v] = 1;
73      } else {
74          // To indicate that a vertex has a loop, we set its
                        label to 1
75          g->label[v] = 1;
76      }
77  }
```

```
78
79  unsigned int read_word(FILE *fp) {
80      unsigned char a[2];
81      if (fread(a, 1, 2, fp) != 2)
82          fail("Error reading file.\n");
83      return (unsigned int)a[0] | (((unsigned int)a[1]) << 8);
84  }
85
86  // Precondition: *g is already zeroed out
87  // returns max edge label
88  void readBinaryGraph(char* filename, graph_t* g) {
89      FILE* f;
90    int i;
91      if ((f=fopen(filename, "rb"))==NULL)
92          fail("Cannot open file");
93
94      unsigned int nvertices = read_word(f);
95      g->n = nvertices;
96      g->label = calloc(g->n, sizeof *g->label);
97      g->adjmat = calloc(g->n, sizeof *g->adjmat);
98      for(i = 0; i < g->n; i++)
99        g->adjmat[i] = calloc(g->n, sizeof *g->adjmat[i]);
100     printf("%d vertices\n", nvertices);
101
102     printf("paolo2");
103     for (int i=0; i<nvertices; i++) {
104         read_word(f);   // ignore label
105     }
106
107     for (int i=0; i<nvertices; i++) {
108         int len = read_word(f);
109         for (int j=0; j<len; j++) {
110             int target = read_word(f);
111             read_word(f); // ignore label
112             add_edge(g, i, target);
113         }
114     }
115   g->degree = calculate_degrees(g);
116     fclose(f);
117 }
118
119 // Precondition: *g is already zeroed out
120 void readLadGraph(char* filename, graph_t* g) {
121     FILE* f;
122     int i;
123     if ((f=fopen(filename, "r"))==NULL){
124         free(g);
125       fail("Cannot open file");
126     }
```

```
127     int nvertices = 0, w;
128     if (fscanf(f, "%d", &nvertices) != 1)
129         fail("Number of vertices not read correctly.\n");
130     g->n = nvertices;
131     g->label = calloc(g->n, sizeof *g->label);
132     g->adjmat = calloc(g->n, sizeof *g->adjmat);
133     for(i = 0; i < g->n; i++)
134       g->adjmat[i] = calloc(g->n, sizeof *g->adjmat[i]);
135     for (int i=0; i<nvertices; i++) {
136         int edge_count;
137         if (fscanf(f, "%d", &edge_count) != 1)
138             fail("Number of edges not read correctly.\n");
139         for (int j=0; j<edge_count; j++) {
140             if (fscanf(f, "%d", &w) != 1)
141                 fail("An edge was not read correctly.\n");
142             add_edge(g, i, w);
143         }
144     }
145   g->degree = calculate_degrees(g);
146     fclose(f);
147 }
148
149 void readGraph(char* filename, graph_t* g, char format) {
150     if (format=='L') readLadGraph(filename, g);
151     else if (format=='B') readBinaryGraph(filename, g);
152     else fail("Unknown graph format\n");
153 }
154
155 graph_t *induced_subgraph(graph_t *g, int *vv) {
156   graph_t * subg = calloc(1, sizeof *subg);
157   subg->n = g->n;
158   subg->label = calloc(g->n, sizeof *subg->label);
159   subg->adjmat = calloc(g->n, sizeof *subg->adjmat);
160   for (int n = 0; n < g->n; n++) subg->adjmat[n] = calloc(g->n,
                         sizeof *subg->adjmat[n]);
161   for (int i = 0; i < subg->n; i++)
162     for (int j=0; j < subg->n; j++)
163       subg->adjmat[i][j] = g->adjmat[vv[i]][vv[j]];
164   for (int i=0; i<subg->n; i++)
165     subg->label[i] = g->label[vv[i]];
166   subg->degree = calculate_degrees(subg);
167   return subg;
168 }
169
170 int graph_edge_count(graph_t *g) {
171     int count = 0;
172     for (int i=0; i<g->n; i++)
173       count += g->degree[i];
174     return count;
```

```
175 }
176
177 void free_graph(graph_t *g){
178   for(int i = 0; i < g->n; i++)
179     free(g->adjmat[i]);
180   free(g->adjmat);
181   free(g->label);
182   free(g->degree);
183   free(g);
184   return;
185 }
186
187 graph_t *sort_vertices_by_degree(graph_t *g, bool ascending ){
188   int *vv = malloc(g->n * sizeof *vv );
189   for (int i=0; i<g->n; i++) vv[i] = i;
190   if (ascending) {
191     INSERTION_SORT(int, vv, g->n, (g->degree[vv[j-1]] >
                        g->degree[vv[j]]))
192   } else {
193     INSERTION_SORT(int, vv, g->n, (g->degree[vv[j-1]] <
                        g->degree[vv[j]]))
194   }
195
196   graph_t *g_sorted = induced_subgraph(g, vv);
197   free(vv);
198   free_graph(g);
199   return g_sorted;
200 }
201
202
203
204
205
206 #endif /* GRAPH_H_ */
```

Bibliography

Barrow, Harry G. and Rod M. Burstall (1976). "Subgraph isomorphism, matching relational structures and maximal cliques". In: *Inf. Process. Lett.* 4.4, pp. 83–84.

Brint, Andrew T and Peter Willett (1987). "Algorithms for the identification of three-dimensional maximal common substructures". In: *Journal of Chemical Information and Computer Sciences* 27.4, pp. 152–158.

Bron, Coenraad and Joep Kerbosch (1973). "Finding all cliques of an undirected graph (algorithm 457)". In: *Commun. ACM* 16.9, pp. 575–576.

Cao, Yiqun, Tao Jiang, and Thomas Girke (2008). "A maximum common substructure-based algorithm for searching and predicting drug-like compounds". In: *Bioinformatics* 24.13, pp. i366–i374.

Cone, Michael M, Rengachari Venkataraghavan, and Fred W McLafferty (1977). "Computer-aided interpretation of mass spectra. 20. Molecular structure comparison program for the identification of maximal common substructures". In: *Journal of the American Chemical Society* 99.23, pp. 7668–7671.

Cook, Shane (2012). *CUDA programming: a developer's guide to parallel computing with GPUs*. Newnes.

De Santo, M. et al. (2003). "A large database of graphs and its use for benchmarking graph isomorphism algorithms". In: *Pattern Recogn. Lett.* 24.8, pp. 1067–1079. ISSN: 0167-8655. DOI: 10.1016/S0167-8655(02)00253-2. URL: http://dx.doi.org/10.1016/S0167-8655(02)00253-2.

Diestel, Reinhard (2018). *Graph theory*. Springer Publishing Company, Incorporated.

Foggia, P., C. Sansone, and M. Vento (2001). "A Database of Graphs for Isomorphism and Sub-Graph Isomorphism Benchmarking". In: -, pp. 176–187.

Garey, Michael R and David S Johnson (2002). *Computers and intractability*. Vol. 29. wh freeman New York.

Hoffmann, Ruth et al. (2018). "Observations from Parallelising Three Maximum Common (Connected) Subgraph Algorithms". In: *International Conference on the Integration of Constraint Programming, Artificial Intelligence, and Operations Research*. Springer, pp. 298–315.

Kann, Viggo (1992). "On the approximability of the maximum common subgraph problem". In: *Annual Symposium on Theoretical Aspects of Computer Science*. Springer, pp. 375–388.

Kerrisk, Michael (2010). *The Linux Programming Interface: A Linux and UNIX System Programming Handbook*. 1st. San Francisco, CA, USA: No Starch Press. ISBN: 1593272200, 9781593272203.

Koch, Ina (2001). "Enumerating all connected maximal common subgraphs in two graphs". In: *Theoretical Computer Science* 250.1-2, pp. 1–30.

Kriege, Nils (2015). "Comparing graphs". PhD thesis. Ph. D. thesis, Technische Universität Dortmund.

Krissinel, Evgeny B and Kim Henrick (2004). "Common subgraph isomorphism detection by backtracking search". In: *Software: Practice and Experience* 34.6, pp. 591–607.

Levi, Giorgio (1973). "A note on the derivation of maximal common subgraphs of two directed or undirected graphs". In: *Calcolo* 9.4, p. 341.

Li, Wenchao, Zach Wasson, and Sanjit A Seshia (2012). "Reverse engineering circuits using behavioral pattern mining". In: *Hardware-Oriented Security and Trust (HOST), 2012 IEEE International Symposium on*. IEEE, pp. 83–88.

McCreesh, Ciaran (2017). "Solving hard subgraph problems in parallel". PhD thesis. University of Glasgow.

McCreesh, Ciaran, Samba Ndojh Ndiaye, et al. (2016). "Clique and constraint models for maximum common (connected) subgraph problems". In: *International Conference on Principles and Practice of Constraint Programming*. Springer, pp. 350–368.

McCreesh, Ciaran, Patrick Prosser, and James Trimble (2017). "A partitioning algorithm for maximum common subgraph problems". In:

McGregor, James J (1982). "Backtrack search algorithms and the maximal common subgraph problem". In: *Software: Practice and Experience* 12.1, pp. 23–34.

Morpurgo, R (1971). "Un metodo euristico per la verifica dell'isomorfismo di due grafi semplici non orientati". In: *Calcolo* 8.1, pp. 1–31.

Nvidia, CUDA (2018a). *H.Compute Capabilities*. `https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html\#compute-capabilities`.

— (2018b). *Programming guide*.

Presa, José Luis López (2009). "Efficient algorithms for graph isomorphism testing". PhD thesis. Ph. D. dissertation, Licenciado en Informatica Madrid.

Raymond, John W and Peter Willett (2002a). "Effectiveness of graph-based and fingerprint-based similarity measures for virtual screening of 2D chemical structure databases". In: *Journal of computer-aided molecular design* 16.1, pp. 59–71.

— (2002b). "Maximum common subgraph isomorphism algorithms for the matching of chemical structures". In: *Journal of computer-aided molecular design* 16.7, pp. 521–533.

San Segundo, Pablo et al. (2013). "An improved bit parallel exact maximum clique algorithm". In: *Optimization Letters* 7.3, pp. 467–479.

Sanders, Jason and Edward Kandrot (2010). *CUDA by example: an introduction to general-purpose GPU programming*. Addison-Wesley Professional.

Schädler, Kristina and Fritz Wysotzki (1999). "Comparing structures using a Hopfield-style neural network". In: *Applied Intelligence* 11.1, pp. 15–30.

Shoukry, Amin and Mohamed Aboutabl (1996). "Neural network approach for solving the maximal common subgraph problem". In: *IEEE Transactions on Systems, Man, and Cybernetics, Part B (Cybernetics)* 26.5, pp. 785–790.

Trimble, James (2017a). *McSplit implementations*. `https://github.com/jamestrimble/ijcai2017-partitioning-common-subgraph/tree/master/code/james-c`.

— (2017b). *McSplit implementations*. `https://github.com/jamestrimble/ijcai2017-partitioning-common-subgraph/tree/master/code/james-c-simplified`.

— (2017c). *McSplit implementations*. `https://github.com/ciaranm/cpaior2018-parallel-mcs-paper/tree/master/james-cpp-parallel`.

— (2017d). *McSplit implementations*. `https://github.com/ciaranm/cpaior2018-parallel-mcs-paper/tree/master/james-cpp`.

Willett, Peter (1999). “Matching of chemical and biological structures using subgraph and maximal common subgraph isomorphism algorithms”. In: *Rational Drug Design*. Springer, pp. 11–38.