

Scuola di Architettura e Ingegneria
Corso di Laurea in Ingegneria e Scienze Informatiche

Accelerating large data modeling for quantum computation with GPUs

Relatore:

**Chiar.mo Prof.
Dario Maio**

Presentata da:

Chiara Varini

Correlatore:

Dr. Marco Gatta

Abstract

The goal of this dissertation is to introduce a new kind of system capable of increasing the speed of the QUBO model creation, by virtue of the paradigms of parallel programming and GPU computing. At a time when the first Quantum Annealers broke on the scene, QUBO model was applied to solve combinatorial optimisation problems. Quantum Annealers are a type of Quantum Computers that take advantage of the natural properties of physical systems, both classical and quantum, in order to find the optimal solution of an optimisation problem described through a minimisation function. The usage of Quantum Computing techniques boosted the problem solution finding so that, at present, the bottleneck is in the creation of the model itself. The project QUBO on GPU (QoG), presented in this dissertation, aims to propose a brand new approach in building the QUBO model exploiting the GPU computation and so obtaining better performances in terms of speed to solve optimisation problems.

First, we present the basics of Quantum Computing and the necessary concepts to understand the principles behind the Quantum Annealing and the quantum computers based on this metaheuristic. Subsequently we will focus on Quantum Annealing and the related D-Wave's Quantum Annealer, the only one existing so far and so highly representative. After this introduction to the world of Quantum Computing, QUBO model is presented by describing it in its mathematical basics and providing two modelling examples: the logic gate AND and the Map Colouring optimisation problem.

An introduction to the General-purpose GPU programming (GPGPU) will then follow, with its main paradigms and architectures and the technology being used in the project, namely CUDA. CUDA is the hardware architecture and framework software that NVIDIA introduced. These two aspects cannot be considered separately but they are inseparable components of the same technology, so CUDA has been used both as a programming language and a GPU architecture.

The main purpose of this work is to create a QUBO model for a generic combinatorial quadratic problem as fast as possible. Since QUBO model is represented via an upper triangular matrix, the project also looks for the best solutions in order to compute and memorise a sparse matrix and how to optimise the access to its entries.

Sommario

L'elaborato di tesi propone un nuovo sistema con il quale è possibile velocizzare la creazione del modello QUBO sfruttando i paradigmi della programmazione parallela e il calcolo su GPU. Questo modello è stato reintrodotta nel campo della risoluzione dei problemi combinatoriali di ottimizzazione grazie alla nascita dei primi Quantum Annealer. Questa categoria di quantum computer sfrutta le naturali proprietà dei sistemi fisici, sia classici che quantistici, per trovare la soluzione ottima di un problema di ottimizzazione descritto tramite una funzione di minimizzazione. Grazie all'utilizzo del quantum computing è stato possibile abbattere i tempi per la risoluzione di tale modello. Tuttavia, per avere un ulteriore vantaggio utilizzando il modello QUBO è necessario anche accelerare la parte di creazione della matrice QUBO relativa al problema di interesse. Tale processo infatti comporta ancora un gravoso rallentamento del modello stesso.

Il progetto QUBO on GPU (QoG), presentato nella tesi, propone un nuovo approccio per la costruzione del modello QUBO che sfrutta la computazione su GPU. Questa soluzione permette di diminuire considerevolmente il tempo utilizzato per questa fase e quindi di ottenere una notevole accelerazione anche nella risoluzione di problemi di ottimizzazione.

Nella prima parte della tesi vengono presentati i concetti fondamentali di quantum computing che stanno alla base del quantum annealing. Viene poi proposto un approfondimento sul Quantum Annealing e, in particolare, sul Quantum Annealer dell'azienda D-Wave poichè, al momento, risulta essere il più rappresentativo. Quindi viene presentato il modello QUBO attraverso una sua descrizione matematica e due esempi applicativi: la modellazione della porta logica AND e la modellazione del problema di ottimizzazione Map Coloring.

Nella seconda parte viene sviluppata un'analisi della programmazione general-purpose su GPU presentandone i principali paradigmi ed architetture. Segue un approfondimento di CUDA la piattaforma hardware e software proposta da NVIDIA. Data la sua duplice natura CUDA viene analizzata sia come architettura hardware che come linguaggio di programmazione.

Questo lavoro si pone come principale obiettivo quello di riuscire a realizzare velocemente il modello QUBO associato a un generico problema combinatoriale quadratico. Essendo il modello QUBO una matrice quadrata superiore, il progetto analizza anche le migliori soluzioni per la memorizzazione di una matrice sparsa e le modalità per ottimizzare l'accesso agli elementi di tale struttura.

keywords: Quantum Computing, Quantum Annealing, QUBO, D-Wave, GPGPU, CUDA

Contents

1	Introduction	13
2	Theoretical Background	17
2.1	Fundamentals of Quantum Computing	17
2.1.1	State	18
2.1.2	Entanglement	20
2.1.3	Quantum gates	21
2.1.4	Tunnelling	24
2.2	Adiabatic Quantum Computing	24
2.2.1	D-Wave	26
2.2.2	QUBO Model	29
2.2.3	QUBO implementing AND logic gate	31
2.2.4	QUBO implementing Map Colouring Problem	33
3	Technical Background	37
3.1	General-purpose computing on GPU	37
3.1.1	Standard architectures	38
3.2	CUDA C	40
3.2.1	Compilation	40
3.2.2	Execution	41
3.2.3	Architecture	43
4	QoG	47
4.1	Requirements analysis	47
4.1.1	Problem definition	48
4.1.2	Constraints definition	50
4.1.3	Building of the QUBO matrix	50
4.1.4	Matrix saving	51
4.2	Design	51
4.2.1	Uploading the problem data	53
4.2.2	Defining the constraints	54
4.2.3	Creating the matrix	54
4.2.4	Saving the data	57
4.3	Technologies	59
4.3.1	Scala	59
4.3.2	CUDA and JCuda	61
4.4	Implementation	64

4.4.1	Scala advanced features	64
4.4.2	Indexing	65
4.5	Testing and performances	68
4.6	Future developments	72
5	Conclusions	73

List of Figures

2.1	Bloch sphere [39]	20
2.2	Quantum circuit to entangle two qubits	22
2.3	Quantum tunnelling	24
2.4	Graphic representation of Quantum Tunneling and Adiabatic evolution [44]	26
2.5	D-Wave QPU [12]	27
2.6	D-Wave Quantum Annealer [12]	27
2.7	D-Wave Quantum cloud service [23]	28
2.8	D-Wave's Ocean SDK stack [23]	28
2.9	QUBO matrix to simulating the AND gate	31
2.10	Coloring a map of Canada with four colors [16]	33
2.11	Problem colouring regions connections graph	34
2.12	QUBO model for colouring regions problem	35
3.1	Many-core vs Multiple-thread models	38
3.2	Heterogeneous co-processing computing model	38
3.3	OpenMP vs MIP architectures	39
3.6	CUDA program simple processing flow [50]	42
3.7	Multidimensional CUDA grid	44
3.8	NVIDIA GPU GeForce Streaming multiprocessor structure	46
4.1	QUBO costs examples	49
4.2	QUBO costs loaded from problem.csv file	49
4.3	QUBO constrained example	51
4.4	System main data flow	52
4.5	System architecture	53
4.6	DataLoader flow	54
4.7	Single constraint definition	55
4.8	QUBO matrix computation	55
4.9	Row-major model	56
4.10	Kernel Factory	57
4.11	KernelDimension Factory	57
4.12	Matrix saving functions	58
4.13	NVCC CUDA compilation	62
4.14	Technological stack	63
4.15	Resources and Job computation	65
4.16	Test coverage	68

4.17 QUBO space usage	69
4.18 Testing QUBO	70
4.19 Test results	71

Listings

2.1	Ocean QUBO example implementation	32
3.1	CUDA memory management	42
3.2	CUDA threads indexing [11]	44
4.1	Type members example [37]	59
4.2	Implicits example [30]	60
4.3	Type members definition	64
4.4	Implicits definition	64
4.5	resources and jobs ids computing	66
4.6	CUDA kernel implementation	66
4.7	CUDA kernel launching	67
4.8	Python script for creating QUBO models	70

Chapter 1

Introduction

Moore's law has been an accurate description of the computers power growth for decades, stating that every 18 months a chip doubles its number of transistors. According with this law, we are now currently dealing with systems (i.e. transistors) just made of particles. Being atomic elements, they do not follow the laws of Classic Physics, but those of Quantum Physics, in particular those of Quantum Mechanics. Born blending Quantum Mechanics and Computer and Information Science, Quantum Computing (QC) is a new branch of Science that focuses on the information processing tasks that can be accomplished using quantum systems. In 1985 British physicist David Elieser Deutsch formally expressed the first Universal Quantum Turing Machine, which is the quantum equivalent of the Classical Turing Machine in "normal" ICT theory and technology.

The QC fundamental data unity is the Quantum Bit, generally called *qubit*. Qubits deeply differ from the common bit idea and provide a quantum counter part. Qubits can be implemented through any double-state quantum system (i.e. the electron spin) and the main differences between qubit and bit come from the typical quantum system phenomenons:

- superposition of state;
- entanglement;
- tunnelling.

All these properties are described in the chapter 2. Thanks to these properties it is possible to boost some specific tasks, which are classically considered computationally hard such as the integer factorisation proposed by Peter Shor in [46], the efficient database research designed by Lov Kumar Grover in [29] and many more. Moreover, Quantum Computing permits also to solve problems that cannot be classically solved. The experiment presented in [49] is the very first try to bring out the *quantum supremacy*.

Two main QC technologies broke into the scene to the present day are reported below:

- Universal quantum computer: built on the gate-based quantum computing model and they handle all the computation phases by controlling the status

changes of the qubits. Universal Quantum Computers can only use some dozen of qubits, given the complexity in managing the interaction among qubits¹;

- Quantum annealer: built on the Adiabatic Quantum Computing (AQC) model also called Quantum Annealing (QA). It exploits the natural evolution of quantum systems and therefore not every bit needs to be checked, it can perform computations on several thousands of qubits².

Thanks to the high number of qubits available to Quantum Annealers, it is possible to speed up systems and to find new solution methods in different fields, such as complex discrete optimisation systems, Artificial Intelligence (AI), Machine Learning and materials science simulations [14]. The solution of an optimisation problem using a Quantum Annealer is made possible by a specific model called Quadratic Unconstrained Binary Optimization (QUBO) and will be detailed in section 2.2. The critical point that this model presents, lies from the performances point of view, that is to say the creation of the model itself. In fact, this aspect requires more time than the other phases of the design work[8].

The main goal of this work is to propose a system capable of creating the QUBO model for any problem as efficiently as possible: the QUBO on GPU system (QoG) presented in the dissertation allows us to define the model in an easy way. The main idea is to massively parallelise the creation of the QUBO model, taking full advantage of GPU computation. This kind of approach permits then to cut the creation timeline and increasing the speed up in solving problems on Quantum Annealers.

The entire work has been completely designed, developed and tested with support from the Data Reply Quantum Computing Team in the premises of Turin. The dissertation is organised as follow:

- **Chapter 1:** the basics of Quantum Computing, Quantum Annealing and QUBO model are presented, so to clarify the goal of the project. The chapter is subdivided into two parts as reported below:
 1. **Fundamentals of Quantum Computing:** firstly, the basic unit of quantum information, namely qubit, is presented with all its properties. The state of a single qubit is then described in mathematical and geometrical terms and then a brief description is given of to the phenomenon of *entanglement* and the different kind of available operators to manipulate the state of one or more qubits that are named *quantum gates*. This part is closed by the presentation of the tunnelling effect, widely used in Quantum Annealers.
 2. **Adiabatic Quantum Computing:** this section gives a quick overview on the state-of-art of Quantum Computers, with a special focus on D-Wave Quantum Annealer. Then it analyses in detail the QUBO model presenting two concrete case studies: the first is purely theoretical and shows how QUBO can be used to simulate the working of the AND gate, while the second one is more realistic and is centred on the solution of a map-colouring problem on a concrete example.

¹At the end of 2019 Google Quantum Computer had 54 qubit

²D-Wave latest model is composed by more than 5000 qubits

- **Chapter 2:** technical issues of general purpose programming on GPU are presented as well as the main architectures available on the market. This chapter is subdivided into two sections as below:

1. **General-purpose computing on GPU:** in this section a brief glance on the evolution and the state of art of GPGPU programming with all the major architectures and framework available for that purpose is given.
2. **CUDA C:** a detailed review on architecture for GPU CUDA and the related API provided from the homonymous framework is presented. Furthermore, the functioning of the dedicated compiler NVCC and how the paradigm SPMD handles the computation is described; finally, the hardware structure CUDA uses is presented with a focus on the addressing of the threads for the parallel computation.

Chapter 3: in this chapter, the QoG project, specially designed for this thesis, is presented through a six sections articulation:

1. **Requirements analysis:** here all the QoG requirements are defined with all the most relevant issues. Each requirement is deeply analysed providing specific examples when needed, so the requirement itself can be better explained.
2. **Design:** in this section the design choices are displayed. Through the use of some UML schemes, the whole architecture is explained, including detailed descriptions of the main components. The inputs and outputs for each phase are highlighted by flow charts, describing also the proper functioning of any part.
3. **Technologies:** here the different technologies that have been used for the development of the system are presented, with the emphasis on Scala advanced features and framework JCuda main properties. It will be shown also how to interface a Java (and so Scala) project to a CUDA project by using JCuda.
4. **Implementation:** in this section the main implementation solutions are displayed, presenting only the most relevant portion of code. Furthermore is reported a detailed description of the QUBO matrix indexing using a row-major model and how acceding to each cell in the smartest way, avoiding useless cycles.
5. **Testing and performances:** the effective functioning of the system is here presented, and for each one of the goal (time and space) detailed tables are given with all performances recorded during appropriate tests brought forth during the validation stage. The execution time is flanked by the hardware features on which the tests were carried out.
6. **Future developments:** in this final part, an overview on the main future developments for QoG is given. Every proposal and blueprint for development is thought to be more and more independent and accessible through other systems. These proposals are only sketched and must be deepened and evaluated during further future development work.

Chapter 2

Theoretical Background

This chapter provides an introduction to Quantum Computing, in order to make the project covered by the thesis understandable to the reader. The chapter is composed by two sections:

- a description of the basics of quantum computing;
- a focus on the Adiabatic Quantum Computing splitting in:
 - glance on the state-of-the-art of the D-Wave system;
 - QUBO model and its usage with the quantum annealers in solving optimisation problems;
 - two examples of modelling with the QUBO.

2.1 Fundamentals of Quantum Computing

The quantum bit, usually shortened to *qubit*, is the fundamental unit of data for quantum computing. As the classical bit, a qubit can be described according to two perspectives, namely from a mathematical or a physical point of view. In this section we will focus mostly on the qubit properties in mathematical terms; some examples of physical implementations of a qubit can be found in [39] and [28]. As we will see, the qubit presents profound differences with its classical counterpart: if a bit can represent one of two different states (0, 1), potentially, a qubit can represent an infinite number of states between 0 and 1. This property is called *superposition of states* and it will be described more in detail in the next section. While a classical bit can be represented by any system that can be in exactly one of the two different states, (i.e. logically a transistor acts as a switch), a qubit must be rather represented by a quantum system, such as the spin of an electron or the polarisation of a photon. Therefore, their behaviour is intrinsically probabilistic and the outcome of measurements of such systems is described in depth by Probability Theory[4].

2.1.1 State

To represent a probabilistic state, mathematically speaking, it is possible use a linear combinations of two vectors. Being a quantum system, the state of a qubit can be also represented by a vector, in particular by a unitary vector in a two-dimensional complex space, that is a Hilbert space. The canonical computational base of a qubit state is $\{(1, 0), (0, 1)\}$ that can be expressed as $\{|0\rangle = \begin{pmatrix} 1 \\ 0 \end{pmatrix}, |1\rangle = \begin{pmatrix} 0 \\ 1 \end{pmatrix}\}$.

The standard notation used to describe a quantum state is the *Bra-Ket* notation, and it was introduced by British physicist Paul Dirac. This particular name comes from the notation itself which is, in fact, mainly composed by the two symbols:

- $\langle \dots |$ called *bra* that corresponds to a horizontal vector;
- $|\dots\rangle$ called *ket* that corresponds to a vertical vector.

As a result, we obtain that the $\langle x | y \rangle$ corresponds to the inner product of the two vectors x, y and the $|x\rangle \langle y|$ their outer product in a finite-dimensional Hilbert space [2].

The main difference¹ between bits and qubits is that a bit can be just in one of two states (0,1), whilst a qubit can be expressed as a linear combination of them, often called superposition [39]. The qubit state $|0\rangle$ and $|1\rangle$ can be seen as the corresponding bit state 0 and 1, but only a qubit can be in a state like:

$$|\psi\rangle = \alpha |0\rangle + \beta |1\rangle$$

In this state, the different members correspond to:

- $|0\rangle$ is the vector $(1, 0)^T$;
- $|1\rangle$ is the vector $(0, 1)^T$;
- α is the *amplitude* of $|0\rangle$;
- β is the *amplitude* of $|1\rangle$;

In order to measure a quantum system we have to interact and perturb it and so changing its state. In Quantum Mechanics this typical effect is explained by the *decoherence* principle. That states that when a quantum system interacts with the environment, it changes its states and loses its quantum properties. Being a quantum system, a qubit follows all the Quantum Mechanics laws too, so it is subject even to the quantum decoherence principle; for a qubit that means when it is observed or measured, its state will collapsed to $|0\rangle$ or $|1\rangle$. This implies that even if a qubit can store a possible unlimited amount of data, these information can not be accessed directly. Measuring a qubit, the probability to get $|0\rangle$ as a result is $|\alpha|^2$ and to get $|1\rangle$ is $|\beta|^2$, and as consequence the fact that $|\alpha|^2 + |\beta|^2 = 1$.

¹A very intuitive and well formed table reporting all the main differences between the qubits and bits is at page 35 of [38]

Moreover the result of a qubit measurement depends on which *base* we choose to do it. Usually the standard base $|0\rangle, |1\rangle$ is used but, in principle, we can resort to any \mathbb{C}^2 orthonormal base to measure a qubit state. Using a different base the measurement result will be one of the two states of the computational base. For example another frequently base used is $|+\rangle, |-\rangle$ in which

$$|+\rangle = \frac{1}{\sqrt{2}}(|0\rangle + |1\rangle) \quad (2.1)$$

$$|-\rangle = \frac{1}{\sqrt{2}}(|0\rangle - |1\rangle). \quad (2.2)$$

Using this last base we can translate the qubit state using

$$|0\rangle = \frac{1}{\sqrt{2}}(|+\rangle + |-\rangle) \quad (2.3)$$

$$|1\rangle = \frac{1}{\sqrt{2}}(|+\rangle - |-\rangle). \quad (2.4)$$

and so obtaining:

$$|\psi\rangle = \alpha|0\rangle + \beta|1\rangle = \frac{\alpha + \beta}{\sqrt{2}}|+\rangle + \frac{\alpha - \beta}{\sqrt{2}}|-\rangle \quad (2.5)$$

Therefore, the subsequent measurement of the qubit will be $|+\rangle$ or $|-\rangle$, both with the same probability $\frac{1}{2}$.

Another possible way to describe a qubit, is by using a geometrical representation. In fact, it is also possible to rewrite a qubit state as:

$$|\psi\rangle = e^{i\gamma} \left(\cos\frac{\theta}{2}|0\rangle + e^{i\phi}\sin\frac{\theta}{2}|1\rangle \right)$$

where θ, ϕ and γ are real numbers. Essentially, they define a point on a unit three-dimensional sphere called *Block sphere* and depicted in fig. 2.1. In this sphere, if we use the standard measurement base z :

- the north pole represents the qubit state $|0\rangle$;
- the south pole represents the $|1\rangle$;

while any other state is a superposition of this two. With this representation any measurement can be seen as a projection of its state onto the computational basis.

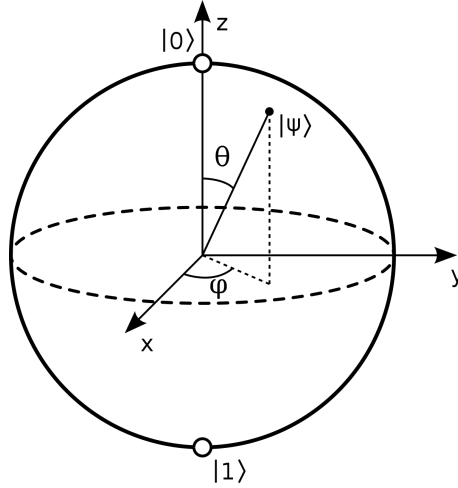


Figure 2.1: Bloch sphere [39]

2.1.2 Entanglement

A quantum system can be composed by multiple qubits so that the whole system state is represented by the tensor product of all them.

The tensor product, represented by \otimes , is an operation that combines vector spaces in order to build a big ones. Given two complex vector spaces \mathbb{C}^m and \mathbb{C}^n and two vectors $v \in \mathbb{C}^m$ and $w \in \mathbb{C}^n$ the tensor product is defined as:

$$\otimes : \mathbb{C}^m \times \mathbb{C}^n \rightarrow \mathbb{C}^{mn} \quad (2.6)$$

with

$$v \otimes w = \begin{pmatrix} v_1 w \\ \vdots \\ v_j w \\ \vdots \\ v_m w \end{pmatrix} \quad (2.7)$$

where for each $1 \leq j \leq m$, $v_j w$ is the product between the column vector w and the scalar value v_j .

So a quantum system composed by two qubits can be:

$$\begin{aligned} |\psi\rangle &= |00\rangle \\ &= |0\rangle \otimes |0\rangle \\ &= (1, 0, 0, 0)^T \end{aligned} \quad (2.8)$$

A system composed by multiple qubit is *entangled* if its state cannot be written as a tensor product of its component. The entanglement is one of the most discussed property of quantum systems. If two or more qubits are entangled, it means they are

very strictly correlated, so, if one of them changes its state, this change instantly modifies the state of the other qubits. Furthermore, if a qubit is measured when two or more qubits are entangled, the state of all the others in entanglement can be found deterministically. The best examples of entanglement are the *Bell states* known also as *EPR pairs*. These are the basic two-qubits entangled systems and each one of them cannot be written as product of its components. They are:

$$\begin{aligned}
 |\beta_{00}\rangle &= \frac{|00\rangle + |11\rangle}{\sqrt{2}} \\
 |\beta_{01}\rangle &= \frac{|01\rangle + |10\rangle}{\sqrt{2}} \\
 |\beta_{10}\rangle &= \frac{|00\rangle - |11\rangle}{\sqrt{2}} \\
 |\beta_{11}\rangle &= \frac{|01\rangle - |10\rangle}{\sqrt{2}}
 \end{aligned} \tag{2.9}$$

The correlations among the qubits in an entangled quantum system finds a lot of useful applications, such as quantum teleportation and super-dense coding [39].

2.1.3 Quantum gates

The quantum gates are used to manipulate the state of a quantum system. A quantum gate is represented by a unitary and subsequently reversible matrix, that performs a change on the qubit state. On the Bloch sphere it can be seen as a rotation of a vector $|\psi\rangle$ representing the qubit state.

The fundamental single-qubit gates are the four *Pauli transformations* (I, X, Y, Z) and the *Hadamard* matrix (H):

$$\sigma_I = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} \qquad \sigma_x = \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix} \tag{2.10}$$

$$\sigma_y = \begin{bmatrix} 0 & -i \\ i & 0 \end{bmatrix} \qquad \sigma_z = \begin{bmatrix} 1 & 0 \\ 0 & -1 \end{bmatrix} \tag{2.11}$$

$$H = \frac{1}{\sqrt{2}} \begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix} \tag{2.12}$$

The Hadamard gate is one of the most important operations to perform on a qubit since it turns its state into a perfect superposition. That means that measuring it, we will have the same probability to get $|0\rangle$ or $|1\rangle$ [43].

A widely used quantum gate on two qubits is the *CNOT* gate.

$$CNOT = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{bmatrix}$$

This gate takes two qubits as input and returns two qubits: if, and only if, the former qubit is in the state $|1\rangle$, the latter flips its state. As shown in the fig. 2.2, the CNOT gate, together with the *H* gate, is used to put two qubits in an entangled state [5]. In the following quantum circuit, replacing x and y with the chosen starting qubits states, we can obtain the same Bell state shown in eq. (2.9).

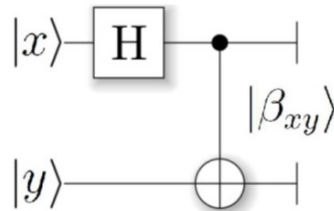


Figure 2.2: Quantum circuit to entangle two qubits

The behaviour of all the explained gates is reported in the table table 2.1 ¹.

¹the gates symbols are taken from [42], page 214

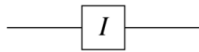
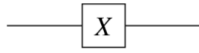
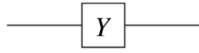
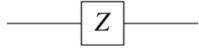
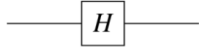
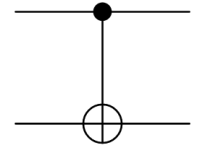
Gate	Symbol	Description
I		It does not change the qubit state. $I 0\rangle = 0\rangle$ $I 1\rangle = 1\rangle$
X		It acts like the classical NOT gate flipping the qubit state. $X 0\rangle = 1\rangle$ $X 1\rangle = 0\rangle$
Y		It flips the qubit state like X and its phase. $Y 0\rangle = i 1\rangle$ $Y 1\rangle = -i 0\rangle$
Z		It flips the qubit phase. $Z 0\rangle = 0\rangle$ $Z 1\rangle = - 1\rangle$
H		It puts the qubit in a perfect superposition. $H 0\rangle = \frac{1}{\sqrt{2}}(0\rangle + 1\rangle)$ $H 1\rangle = \frac{1}{\sqrt{2}}(0\rangle - 1\rangle)$
CNOT		It changes the second qubit state if the first qubit is $ 1\rangle$. $CNOT 00\rangle = 00\rangle$ $CNOT 01\rangle = 01\rangle$ $CNOT 10\rangle = 11\rangle$ $CNOT 11\rangle = 10\rangle$

Table 2.1: Quantum gates

2.1.4 Tunnelling

In Classical Mechanics, a typical object cannot pass through a barrier but it has to go around it to cross it. In the quantum world, however, this is technically made possible by virtue of Heisenberg Uncertainty principle [6]. According to this, a certain subatomic particle does not have a precised state but, in general, it can behave as particle or wave. So in Quantum Mechanics a subatomic particle can pass through a potential barrier with a certain probability, dependent on the barrier weight, thanks to its wave properties. This important property applies in Quantum Annealing to explore an energy landscape and consuming the minimum energy. This last concept will be further explained in the next section. In fig. 2.3, a typical scheme of the tunnelling effect is reported.

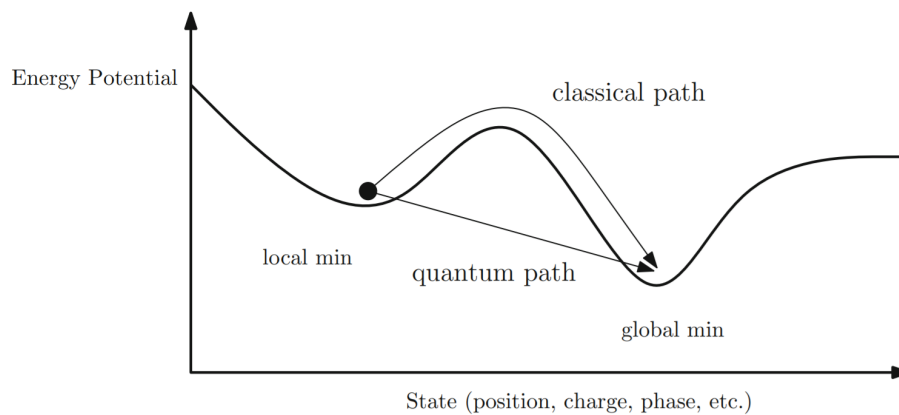


Figure 2.3: Quantum tunnelling

2.2 Adiabatic Quantum Computing

In the field of Quantum Computing, two main technologies took over:

- **quantum gate model:** also known as *circuit model*, this model has been conceived in 1989 by Deutsch's as an universal gold standard when speaking Quantum Computing, and in fact it is still the standard;
- **quantum annealing model:** introduced in 1988 as a *quantum stochastic optimisation*, namely a solver for combinatorial and optimisation problems, it was later renamed Adiabatic Quantum Computing or quantum annealing [48].

Adiabatic Quantum Computing, also known as Quantum annealing, meaningfully differs from the gate model:

- in the gate model the computation is handled as a series of quantum logic gates to apply to a set of qubits initialized to the state $|0\rangle$ and it takes care of all the single transformations of their state;

- in QA the process starts with a system description using a Hamiltonian. Then system is left to evolve until it reaches its ground state that encodes the problem solution¹.

In fact, AQC is a computation model based on the adiabatic theorem of Born-Fock, which describes the natural evolution of a Hamiltonian system towards its lowest energy point. In nature, both in classical and quantum mechanics, the following rules apply: hot things cool down, sloping objects descend and so on. Atoms too tend to their balance point that, in this case, is the one with minimum energy level. The time evolution of a quantum system can be described by Schrödinger equation:

$$i\hbar \frac{d}{dt} |\Psi(t)\rangle = \hat{H} |\Psi(t)\rangle \quad (2.13)$$

where i is the imaginary unit, \hbar is Planck constant, $|\Psi\rangle$ is the state vector of the quantum system and \hat{H} is the Hamiltonian operator. The eigenstate of the Hamiltonian related to the smallest eigenvalue represents the lowest energy system, namely the *ground state*. AQC focuses on Hamiltonian operators and how using them to encode the goal functions of optimisation problems, whose ground state corresponds to the optimal solution for the problem under consideration.

Computers that resort to this kind of approach are known as Quantum Annealers: by virtue of the intrinsic properties of quantum systems, QA can efficiently find optimal (or very good) solutions for combinatorial optimisation problems described by a Hamiltonian. The stages behind this approach are:

1. defining the QA initial state through a known-value ground state H_0 ;
2. defining a goal function using a specific Hamiltonian H_1 ;
3. slowly enabling the system to evolve adiabatically until it reaches H_1 in accordance with

$$H = (1 - s)H_0 + sH_1$$

The parameter s defines the velocity gradient when evolving from H_0 to H_1 . s varies between 0 and 1 and for each change, the corresponding ground state H is evaluated. The less the algorithm cools down in terms of speed, the higher will be the probability to find the lowest energy point of the final system H_1 , that is the initial goal function. This process is reported in fig. 2.4.

However, reaching the ground state cannot be taken for granted: if a local minimum point is achieved we obtain only an approximate solution. The Quantum Annealer also makes use of the quantum properties of the system in order to find better solution in a shorter time. The main properties used at this purpose are:

- superposition: to examine many solutions at once;
- tunneling: to bypass local minimum points without loss of energy, as reported in fig. 2.4;

¹A deeply detailed introduction to AQC can be found in [48]

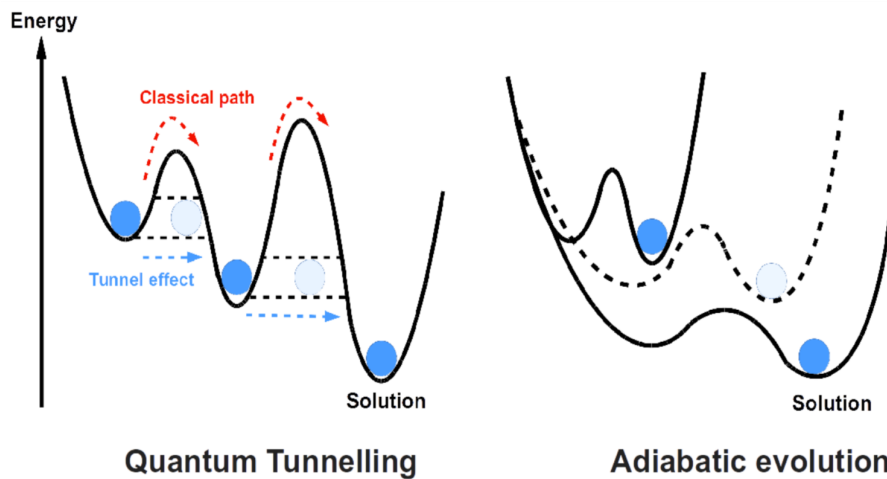


Figure 2.4: Graphic representation of Quantum Tunneling and Adiabatic evolution [44]

- entanglement: to find better solution using the correlations among the minimum points;
- quantum fluctuations: to create particle-antiparticle pairs of virtual particles.

2.2.1 D-Wave

D-Wave is a leader company in developing quantum annealers. Since 1999, it has always proposed cutting-edge designs, until reaching the ongoing 4th generation of Quantum Computers, doubling the qubit numbers with every new model ¹. The process unit of the D-Wave systems is called Quantum Process Unit (QPU) and it is composed by several qubits connected by couplers as depicted in the fig. 2.5. The QPU is designed to solve quadratic unconstrained binary optimisation problems, where each qubit represents a variable, and couplers between qubits represent the costs associated with qubit pairs. The QPU is a physical implementation of an undirected graph with qubits as vertices and couplers as edges among them [40].

To solve an arbitrarily posed binary quadratic problem directly on a D-Wave system it is necessary perform a mapping, called *minor embedding*, to a precise topology that represents the system's quantum processing unit, this procedure is detailed in [17].

The main requirements of the D-Wave's quantum annealer are:

- Cryogenic temperatures: in order to kept the QPU temperature near absolute zero. It is achieved using a closed-loop cryogenic dilution refrigerator system. The QPU operates at temperatures below -273.135° ;
- Shielding from electromagnetic interference: in order to kept the system isolated from the surrounding environment and behave quantum mechanically. It is

¹D-Wave One (2011) operates on a 128-qubit chipset; D-Wave Two (2013) works with 512 qubits, D-Wave 2X (2015) works with 1000 qubits, D-Wave 2000Q (2017) works with 2000 qubits

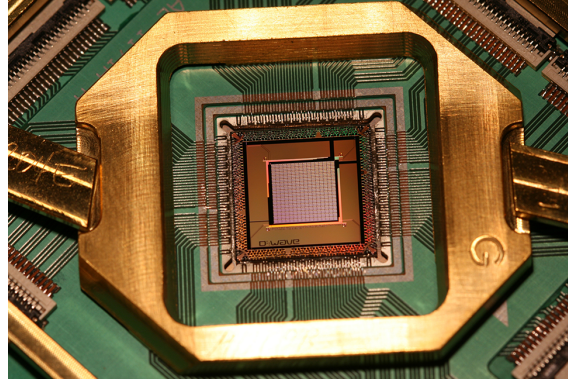


Figure 2.5: D-Wave QPU [12]

achieved using a radio frequency (RF)-shielded enclosure and a magnetic shielding subsystem.



Figure 2.6: D-Wave Quantum Annealer [12]

The current on sale model is called *D-Wave 2000Q* and consists of 2000 qubits. We can see it in fig. 2.6. Every D-Wave model can be directly purchased or remote accessed through the cloud service *Leap*. Both *Leap* and the embedded SDK *Ocean*, are made by D-Wave itself, as represented in fig. 2.7. At the end of 2019, D-Wave announced its next generation model *Advantage* [13], with the following enhancements:

- increasing the qubits on the QPU: *Advantage* will use more than 5000 qubits;
- presenting a brand new topology for qubits: the current *Chimera* gives way to *Pegasus*, rising in the qubits link from 6 to 15¹;
- a noise reduction on the QPU.



Figure 2.7: D-Wave Quantum cloud service [23]

At the present time, Leap is the only cloud service and Quantum Application Environment (QAE): the developing of quantum applications becomes faster and fully supported at an industrial level, supplying a live access to the quantum computer, software development kit (SDK), resources and also to an on-line community. In addition to the cloud service, D-Wave developed also a set of open-source Python tools that, as in classical computer science, simplifies the creation of application without knowing the physical properties of the quantum computer. All these tools are enclosed in D-Wave open-source Ocean SDK. The stack software implements every necessary operation to transform high-level problems into quantum solvable problems.

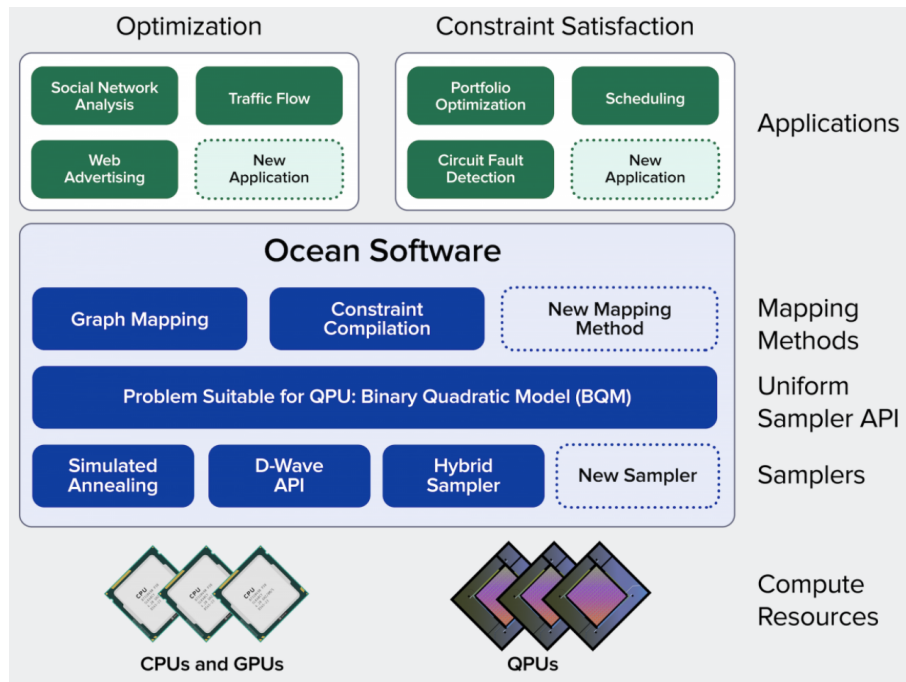


Figure 2.8: D-Wave's Ocean SDK stack [23]

As is clear from fig. 2.8, Ocean is located between the application code and the HW resources. The main features this framework offers, as stated in D-Wave official documentation[1], are as below:

¹More details on both architecture can be found in [35] and in [21]

- Application: original problem in its context (the “problem space”) including application data and a clearly defined goal;
- Mapping Methods: tools that translate the application goal and data into a problem form suitable for quantum computing. They also receive solution samples and translate them back into solutions for the application layer;
- Uniform Sampler API: abstraction layer that represents the problem in a form that can access the selected sampler;
- Samplers: tools that receive a problem in the form of a binary quantum model (BQM) and return solution samples. Ocean implements several samplers that use the D-Wave QPU as well as classical compute resources. You can use Ocean tools to customise a D-Wave sampler, create your own, or use existing classical ones;
- Compute Resources: the processing hardware on which the problem is solved. This might be a D-Wave QPU but it may also be the CPU of your laptop computer.

By virtue of this framework, every user can implement its own application at a very high-level and the next step will be automatically carried out by assigning proper weights to any qubit and to strength of the couplers, namely the links, or arches, between two different qubits. At this point, these values and possible user-defined parameters, are taken as inputs so that the system sends a single Quantum Machine Instruction (QMI) to the QPU. The solutions we obtain are the lowest points in the energy landscape and the number of these solutions can be defined by the user itself. It is important to emphasise that we can obtain more than one solution, because of the probabilistic nature of quantum computing so that good solution may come together the optimal one [20].

The current family of D-Wave computers can solve problems formulated in either Ising form or QUBO form as shown in [36].

2.2.2 QUBO Model

QUBO, an acronym for Quadratic Unconstrained Binary Optimisation, is the most widespread model in solving optimisation problems through quantum annealers. This kind of problems are very common in computer science and include some of the most famous NP-hard problems, namely those problems for which there are not yet efficient algorithms to solve them. That means that real-world problems with a high number of variables cannot be solved with this model using classical machines. However, thanks to the new quantum computing techniques, i.e. quantum annealing, we can find an optimal solution for this problems in short span of time. Some of the most famous real problems which this model can be applied to are: Knapsack [7], traffic flow optimization [40], Max Cut problem [47], Distributed Task allocation problem, Number Partitioning problem and many more [27]. In order to solve a problem using a quantum annealer, it is possible to use another model too, called Ising model. The Isign model represents a natural physics system. The goal is the

same of the QUBO, that is to say: finding the optimal solution in order to minimise an objective function. These two models are very similar, the main difference is the coding of the variables: in the QUBO the variables can be 0 or 1, in the Ising 1 or -1. The conversion between these coding can be done as ¹:

- from QUBO to Ising: $y = s + 1/2$;
- from Ising to QUBO: $s = 2y + 1$;

Ising and QUBO models are exchangeable, as detailed in [15], [51] and [19]. In this section we will focus only on the QUBO, the model chosen for the project.

QUBO allows to solve a lot of combinatorial problems in both private and public industrial fields, where yes or no decisions are required: these decisions all represent different values of the goal function. QUBO permits to solve a lot of these problems in a very efficient way, once they have been formulated following its specific rules.[3]. The term unconstrained does not imply that problems cannot present constraints, but that they are directly put into the model itself, which is the matrix Q . Given a problem, this model is used to find the optimal variables assignment, so that the specific goal function can be minimised. The variables are binary values so they can be only:

- 1, if they are part of the solution;
- 0, otherwise.

The variables assignment is represented by a binary vector $x = (x_0, x_1, \dots, x_{n-1})$, where n is the number of boolean variables the problem presents. If a certain i -th variable is contained in the final solution, the i -th cell of vector x will be a 1, 0 otherwise.

QUBO model is defined by:

- an upper triangular $N \times N$ real numbers matrix Q , where $N = n$;
- a binary variables vector x that minimise the goal function:

$$f(x) = \sum_i Q_{i,i}x_i + \sum_{i<j} Q_{i,j}x_ix_j \quad (2.14)$$

In a QUBO matrix, the entries on the main diagonal $Q_{i,i}$, i represents linear coefficients, while the others above it $Q_{i,j}$ with $j > i$ are the quadratic one. In several studies, in order to underline that the goal is to find the x vector, a more concise notation is preferred such as:

$$\min_{x \in \{0,1\}^n} x^T Q x \quad (2.15)$$

There is also a scalar version of this model, as defined in [23]:

¹ y represents the QUBO variable, s the Ising one

$$E_{qubo}(a_i, b_{i,j}; q_i) = \sum_i a_i q_i + \sum_{i<j} b_{i,j} q_i q_j \quad (2.16)$$

where a_i are the biases or weights for each qubit, q_i , and $b_{i,j}$ the coupler strength between qubits i and j . The optimisation problems are solved by setting biases a_i and coupler strengths $b_{i,j}$ such that the qubits q_i in the minimised objective satisfy the problem constraints. The constraints to be applied to the QUBO model are usually in the form of:

- **equalities:** directly mappable into QUBO matrix [22];
- **inequalities:** not directly mappable into QUBO matrix, in fact we need to use some ancilla variables to represent the *slack* values [19]. An example can be found in the QUBO model applied to a knapsack problem, as reported in [7].

2.2.3 QUBO implementing AND logic gate

To model a problem, an easy way to familiarise with using QUBO, is the one that follows: suppose we want to simulate the logic gate *AND* through a matrix. To do so we need to build a 4×4 matrix because *AND* is a binary operator and all the possible variable combinations are 4 (00, 01, 10 and 11). In this matrix, appropriate weights are to be used so that the final result is the combination of input variables (1, 1), corresponding to the x vector (0, 0, 0, 1). Each value of the vector x corresponds to a QUBO row, so if the i -th x value is 1 the i -th QUBO row is activate and it concurs to the final energy value. QUBO is essentially a matrix whose entries define the initial energy landscape of the quantum annealer and, using quantum annealing techniques, it spans this landscape looking for the optimal solution, namely the one with lesser energy. Given a point, greater is its value (the energy relating to that point) and lesser is the is probability to include that combination in the final solution. On the basis that the QUBO matrix is upper triangular, we ought to decrease the value in the right variable combination, that is 11, and therefore increase the others of a given value, like $\lambda = +10$ as reported in fig. 2.9.

	00	01	10	11
00	0	0	0	0
01	0	0	0	0
10	0	0	0	0
11	0	0	0	0

	00	01	10	11
00	+	+	+	+
01	0	+	+	+
10	0	0	+	+
11	0	0	0	-

	00	01	10	11
00	10	10	10	10
01	0	10	10	10
10	0	0	10	10
11	0	0	0	-10

Figure 2.9: QUBO matrix to simulating the AND gate

All the existing solutions for this problem are presented in table 2.2. As we can see, the desired solution is the one with the lowest value. The costs of each solutions is computed as eq. (2.15) where the x is the selected solution and the Q is the matrix of fig. 2.9 where x is the current solution and Q is the matrix proposed in fig. 2.9.

solution	value	solution	value
0000	0	1000	10
0001	-10	1001	10
0010	10	1010	30
0011	10	1011	40
0100	10	1100	30
0101	10	1101	40
0110	30	1110	60
0111	40	1111	80

Table 2.2: QUBO solutions

This sample can be implemented using the D-Wave framework Ocean ¹ as in the listing 2.1.

```

1
2 from D-Wave_qbsolv import QBSolv
3 Q = {(0, 0): 10, (0, 1): 10, (0, 2): 10, (0, 3): 10,
4      (1, 1): 10, (1, 2): 10, (1, 3): 10,
5      (2, 2): 10, (2, 3): 10,
6      (3, 3): -10}
7
8 response = QBSolv().sample_qubo(Q)
9
10 print("samples=" + str(list(response.samples())))
11 # output = samples=[{0: 0, 1: 0, 2: 0, 3: 1}]
12
13 print("energies=" + str(list(response.data_vectors['energy'])))
14 #output = energies=[-10.0]
```

Listing 2.1: Ocean QUBO example implementation

Using Ocean to solve a QUBO matrix, we firstly need to describe it through a Python dictionary, in which the keys represent the row-column coordinate of the cell in which the default value is to be put in. After that, it is sufficient to recall the function `Qbsolv().sample_qubo(Q)` with the newly created dictionary. The parameter `solver` for the method `sample_qubo()` may be of the following types:

- **'tabu'**(default): sub-problems are called via an internal call to tabu;
- **'dw'**: sub problems are given to the D-Wave library;

¹The Ocean documentation is on the web page [18]

- instance of a **dimod sampler**: the specific `sample_qubo` method is invoked;
- **callable**: that has the signature (`qubo: dict, current_best: dict`) and returns a result list/dictionary with the new solution.

The previous call generates a solutions vector held in the `samples` field, containing all the same energy optimal solutions.

Two interesting examples of QUBO models applied to solve graph covering Isomorphism have been reported in [31].

2.2.4 QUBO implementing Map Colouring Problem

Another interesting problem that can be modelled by QUBO is the map colouring. This problem requires to find the assignment of a colour to each region given a finite set of colours and a finite set of regions. Moreover the colour assignment has to respect the following rules:

- one and only one colour of the set must be assigned to any region;
- a certain colour, once it has been assigned to a region, cannot be used in any other adjacent regions.

An example of a solution for this kind of problem is represented in fig. 2.10.



Figure 2.10: Coloring a map of Canada with four colors [16]

These problems can be modelled as satisfiability problems as follows: let $x_{ij} = 1$ if node i is assigned color j , and 0 otherwise. Since each node must be coloured with only one color, we have the equality:

$$\sum_{j=1}^k x_{ij} = 1 \quad i = 1, \dots, n$$

where k is the number of colours and n is the number of regions. In order to model the feasible colouring constraint, in which different colours are assigned to any adjacent nodes, we can use the following inequality:

$$x_{ip} + x_{jp} \leq 1 \quad p = 1, \dots, k$$

for all adjacent nodes i, j in the graph.

Let us consider the problem of finding a feasible colouring of the fig. 2.11 graph using $K = 3$ colours and $n = 5$, giving a model with 15 variables (5×3), where a single variable x_{ij} corresponds to a one relation between the region i and the colour j .

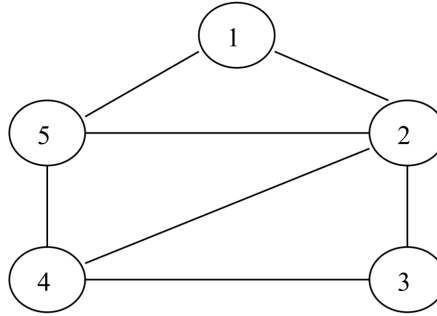


Figure 2.11: Problem colouring regions connections graph

One possible way to proceed is to start with a 15-by-15 Q matrix where initially all the elements are equal to zero and then re-define appropriate elements based on the penalties obtained from the constraints:

$$x_{i1} + x_{i2} + x_{i3} = 1 \quad i = 1, 5 \quad (2.17)$$

$$x_{ip} + x_{jp} \leq 1 \quad p = 1, 3 \quad (2.18)$$

fixing $\lambda_1 = 4$ and $\lambda_2 = 2$ respectively for the former and the latter constraints the resulting QUBO is represented in fig. 2.12. The whole mathematical procedure is detailed in [27]. The constraint expressed by the eq. (2.17) is represented by the blue values while the other expressed by eq. (2.18) by the green values.

	x_{11}	x_{12}	x_{13}	x_{21}	x_{22}	x_{23}	x_{31}	x_{32}	x_{33}	x_{41}	x_{42}	x_{43}	x_{51}	x_{52}	x_{53}
x_{11}	-4	4	4	2	0	0	0	0	0	0	0	0	2	0	0
x_{12}		-4	4	0	2	0	0	0	0	0	0	0	0	2	0
x_{13}			-4	0	0	2	0	0	0	0	0	0	0	0	2
x_{21}				-4	4	4	2	0	0	2	0	0	2	0	0
x_{22}					-4	4	0	2	0	0	2	0	0	2	0
x_{23}						-4	0	0	2	0	0	2	0	0	2
x_{31}							-4	4	4	2	0	0	0	0	0
x_{32}								-4	4	0	2	0	0	0	0
x_{33}									-4	0	0	2	0	0	0
x_{41}										-4	4	4	2	0	0
x_{42}											-4	4	0	2	0
x_{43}												-4	0	0	2
x_{51}													-4	4	4
x_{52}														-4	4
x_{53}															-4

Figure 2.12: QUBO model for colouring regions problem

Chapter 3

Technical Background

The purpose of this chapter is to introduce the basic concepts of heterogeneous systems and the GPU programming with a focus on the framework CUDA. An insight on these ideas will give a better understanding of the technological choices and the implementation details presented in the Project chapter. The first section is focused on the GPGPU programming model while the second on a detailed analysis on CUDA from both hardware and software perspective.

3.1 General-purpose computing on GPU

CPUs and GPUs, although they both execute programs, are a world apart in their design goals: while CPUs use a Multiple Instruction, Multiple Data (MIMD) approach, GPUs use a Single Instruction, Multiple Thread (SIMT) instruction model [9]. GPUs' development has always been the core business of the main video-games industries, whose technological efforts are always made to get with the times in providing more and more realistic virtual experiences. Reaching this goal requires an impressive number of calculations (i.e. rotation-translation), that typically consist in floating point operations for any pixel of the screen. GPUs were precisely created as specific devices conceived for highly parallel computations, which is the case of graphic rendering. GPU's architectures are shaped on multi-thread model rather than many-core, so they are focused on data processing optimisation rather than on data caching and flow control. Both models are represented in fig. 3.1.

The General-purpose on graphical process unit computing paradigm permits to use the GPU computing power not only for the rendering of videogames but also for scientific and engineering generic purpose computation, such as the optimisation in the simulation of real physical systems. Thanks to the GPGPU, data transfer between CPU and GPU becomes bi-directional and, subsequently, systems that require a lot of precise operations to be quickly evaluated on a large amount of data are definitely enhanced in their performances. Any smart GPU implementation of a SIMD can present up to 100x speed up if compared to a sequential one on a single core CPU. According to Flynn's taxonomy, a system is a SIMD (Single Instruction Multiple Data) if a single instruction, or even a small set of instructions, is run in parallel on a lot of data [41].

A generic system is not usually parallelizable in all its parts, so the speed up is

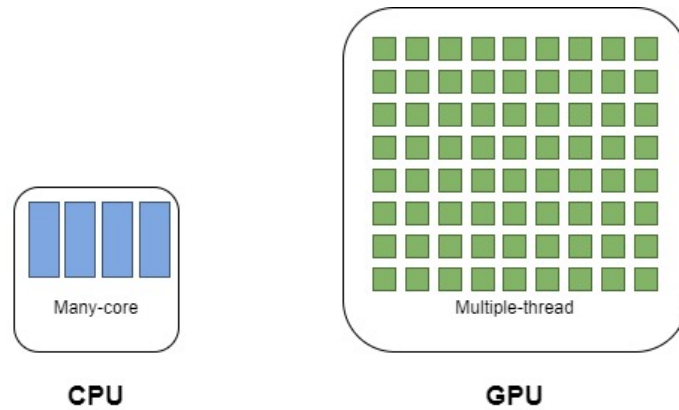


Figure 3.1: Many-core vs Multiple-thread models

normally limited only to a small portion of code. Thus, we need to design systems in which some parts are to be conceived in parallel, while others in a serial way. For that reason, under the GPGPU model, CPU and GPU co-operate with each other, giving rise to a heterogeneous co-processing computation model, in which:

- the CPU is responsible of the sequential part;
- the GPU takes care of the computationally-intensive part.

This duality is clarified in fig. 3.2.

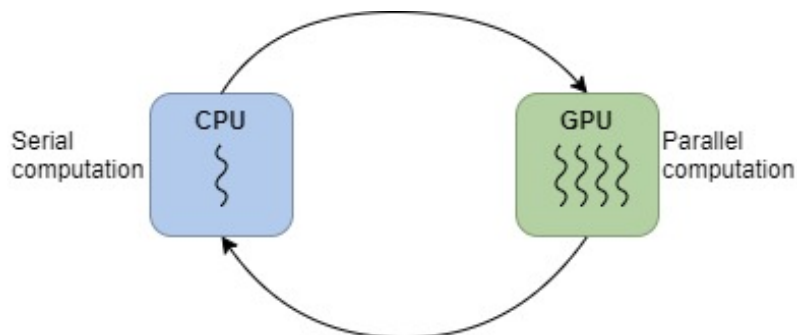


Figure 3.2: Heterogeneous co-processing computing model

At the beginning, GPUs and parallel programming languages were limited to a very diverse market than CPUs. In the "classical" CPU programming, the compatibility among different versions of the same software is, from the beginning, a basic requirement, whilst the innovation in terms of GPU improvement led often to drastic changes in the hardware. Such changes in technology have meant a loss of portability among the different models: because of the new hardware introduction, brand new GPU architectures have been proposed, with significant differences among them. These architecture required almost always a whole re-definition of their codes, then.

3.1.1 Standard architectures

Two different standards have been proposed, in order to avoid the risks mentioned above:

- **OpenMP**: this standard is centred on parallelisation on a single node for memory-shared multi-core machines. The software must run on the same machine, providing an almost easy but limited model to adopt;
- **MPI**: this standard provides for parallelisation on diverse nodes and it is thought to be applied on networked machine clusters. It is often used as a supercomputer architecture, in which thousands of nodes are connected together through a dedicated network. Any problem is split into several sub-parts, each one of them is solved by a specific node. This model is a lot more flexible than the previous one and it requires and offers more resources: in fact, any node keeps its own internal resources (that is to say CPU, cache, storage, etc.). The main limitations for this kind of approach are augmented complexity and network speed, in terms of the mutual interchanges and communications among the nodes.

In fig. 3.3, two representations for these architectures are given.

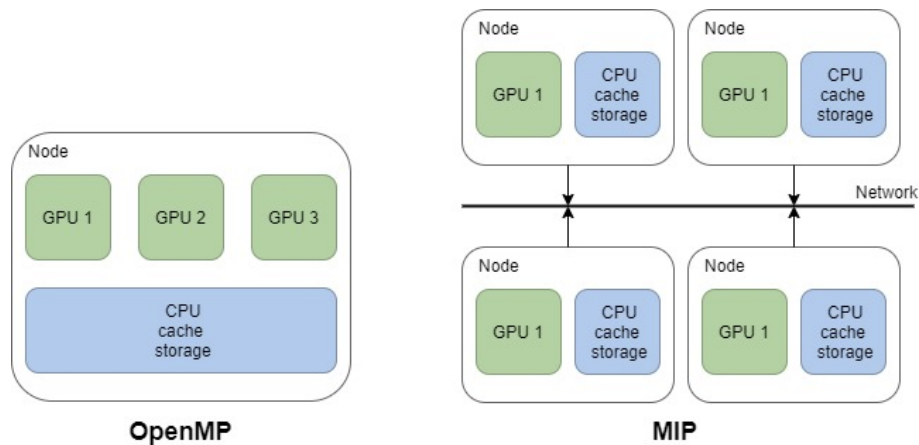


Figure 3.3: OpenMP vs MPI architectures

OpenMP allows the programmer to reach a high-level of parallelisation, specifying the specific portions of code to optimise. On the other hand, MPI explicitly uses the communication among processes in order to increase the amount of work. Due to the intrinsically different nature of these standards, they are almost never used at the same time.

NVIDIA introduced a programming language called CUDA, in order to provide an heterogeneous programming framework. This is a very powerful tool since it allows to use OpenMP and MPI together. In fact, the acronym CUDA stands for Compute Unified Device Architecture. However, CUDA consists also in a hardware architecture so, in order to properly use this framework, a CUDA-capable GPU is needed. Since CUDA has been developed by NVIDIA, each one of its last generation GPU supports it. The architecture of a typical NVIDIA CUDA-capable GPU is shown in fig. 3.4. The differences among the several models can be traced in Streaming Multiprocessors (SMs) and Stream Processors (SPs), but we will not take this concept any further. Each GPU currently comes with gigabytes of Graphics Double Data Rate (GDDR), Synchronous DRAM (SDRAM), referred as Global Memory [24].

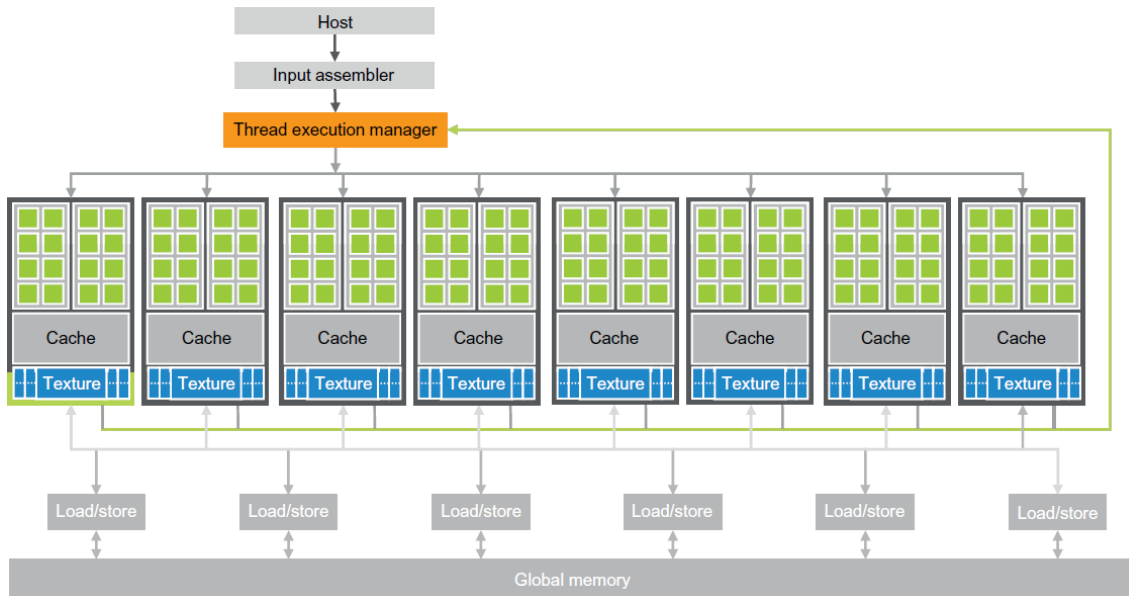


Figure 3.4: Architecture of a CUDA-capable GPU. Image taken from [24] page 20

3.2 CUDA C

As we said before, the acronym CUDA stands for Compute Unified Device Architecture and it is the commercial name for the framework that NVIDIA designed for GPGPU application programming. It provides several extensions of mainstream programming languages such as C, C++, Python, Fortran, OpenCL, OpenACC, OpenMP and many others [24]. In this way it is possible to raise the level of GPU programming by simplifying the implementation of parallel and scalable systems. In fact, this language integration allows that device functions, which must perform on the GPU, look very much like host functions, that have to be performed on the CPU. At run-time the CUDA compiler takes care of the business of invoking device code from the host [32]. This framework is based on a variant of the SIMD model called Single Program Multiple Data model (SPMD), in which all the threads of the GPU run the same CUDA code. Parallel programming is based on the idea of thread, that is the representation of a single execution flow in a program. When paralleled, any thread of the program co-operates with the others to reach the common goal that has been defined in the program itself.

3.2.1 Compilation

CUDA C is an extension made for the programming language C by adding the necessary syntax to implement heterogeneous systems. This way of modelling mirrors the very nature of programming GPGPU, providing the appropriate instructions to handle the consistency of a single host (CPU) and of one or more devices (GPUs) on a specific machine. In fact, any CUDA source code can contain a mixture of both host and device code. To properly compile a CUDA file, it is necessary to use NVCC, the dedicated compiler, that has to split the code into two parts: one for the CPU portion and one for the GPU part. NVCC operates this way:

- the C code is compiled by a classic C/C++ compiler (i.e. GCC);
- the GPU functions, properly marked with specific keywords, are compiled by a just-in-time CUDA compiler.

In fig. 3.5 is shown the architecture of the compiler NVCC.

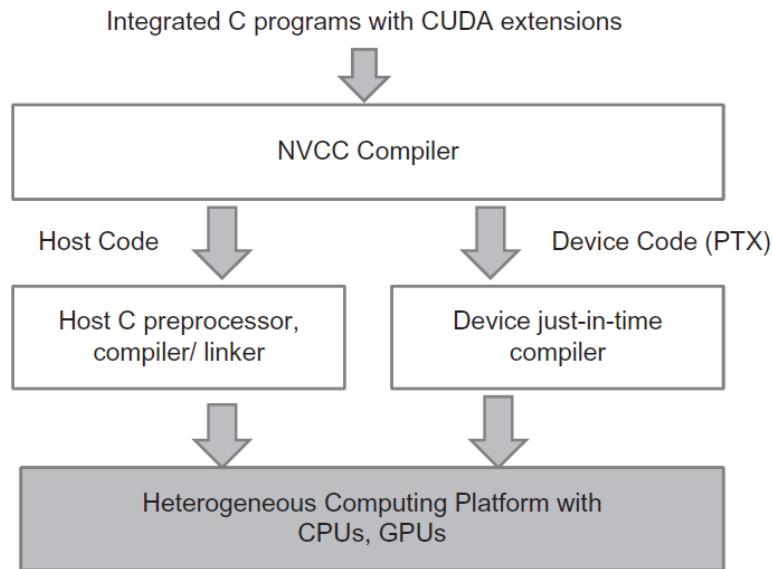


Figure 3.5: NVCC compilation process. Image taken from [24] page 36

In order to distinguish and so detect the host code from the device one, CUDA makes available 4 main keywords:

- `__global__`: the function, started by the host and launched on the device, is a kernel and has to be run on the GPU;
- `__device__`: the function, started by the device, can only be used on the device;
- `__host__`: it marks a C function that is started by the host and launched on it. Every default CUDA functions are `__host__` function so thy keyword is often omitted;
- `__host__ __device__`: this function is used by both the host and the device. This keyword allows the programmer to avoid useless code repetitions.

3.2.2 Execution

Runtime for a CUDA program, starts with the host code launching: when a kernel is started, the control flow is passed to the GPU that will execute the kernel on more threads. Once the parallel computation is ended, the flow control returns under the host control. CUDA allows also to run host code and device simultaneously, but the approach we choose is the most typically used [25]. A specific kernel function specifies the code that has to be run in parallel on the device. In order to run a

kernel on the GPU, we need first to allocate the global memory on the GPU and to transfer them from the host to the GPU: this is a fundamental step because the CPU memory is kept separated from the GPU one. The data acquisition, once the execution of the kernel is over, requires to perform the reverse operation, that is to transfer the data from the GPU to the CPU. At this point, it will be possible to free the GPU memory. A simple example of a processing flow for a CUDA program is shown in fig. 3.6.

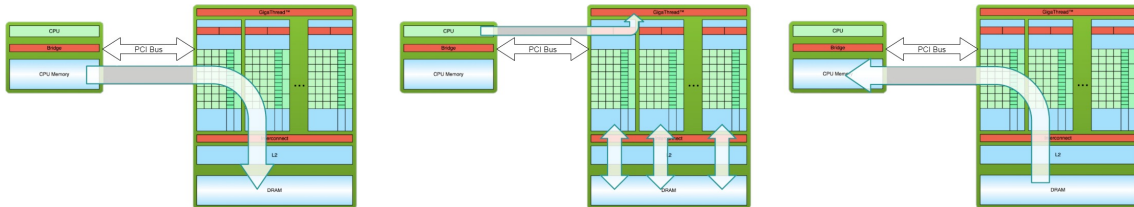


Figure 3.6: CUDA program simple processing flow [50]

Accessing to memory is one of the most critical operation in terms of the system performances. To cope with this, CUDA gives an explicit control on memory allocation and manipulation, providing for the right functions to exchange data between the CPU and the GPU. These functions are reported in the sample code listing 3.1. The standard used to define the variables states that the variables are to be preceded by prefixes:

- $h_$ for the variables on the host;
- $d_$ for the variables on the device.

```

1
2 ...
3
4 int n = 10;
5 int size = n * sizeof(int);
6 int *h_A = (int *) malloc(n*sizeof(int));
7
8 //initialize h_A with some values
9
10 cudaMalloc((void **) &d_A, size);
11 cudaMemcpy(d_A, h_A, size, cudaMemcpyHostToDevice);
12
13 kernel<<<1, 10>>>(args);
14
15 cudaMemcpy(h_A, d_A, size, cudaMemcpyDeviceToHost);
16
17 cudaFree(d_A);
18
19 ...

```

Listing 3.1: CUDA memory management

The `kernelName<<<...>>>()` function can be used to launch a kernel. It takes a configuration that is specified by inserting an expression of the form

<<<Dg, Db, Ns, S >>> between the function name and the parenthesized argument list, where:

- **Dg**: is of type `dim3`¹ and specifies the dimension and size of the grid, such that `Dg.x * Dg.y * Dg.z` equals the number of blocks being launched;
- **Db**: is of type `dim3` and specifies the dimension and size of each block, such that `Db.x * Db.y * Db.z` equals the number of threads per block;
- **Ns**: is of type `size_t` and specifies the number of bytes in shared memory that is dynamically allocated per block for this call in addition to the statically allocated memory; this dynamically allocated memory is used by any of the variables declared as an external array as mentioned in `__shared__`; Ns is an optional argument which defaults to 0;
- **S**: is of type `cudaStream_t` and specifies the associated stream; S is an optional argument which defaults to 0.

There are other useful functions that CUDA provides in order to manage the GPU, such as defining the available GPUs on the machine, how much memory is allocated on each one of them, managing the communication and synchronization among the different kernels, etc.

3.2.3 Architecture

When a program launches a CUDA kernel calling the function `kernelName<<<grid.x, grid.y, grid.z, block.x, block.y, block.z>>>(params);`, the CUDA runtime system creates a threads grid split into blocks. Each block can contain up to 1024 threads and the dimension of the grid and thread blocks are defined into the function with which the kernel is run within the 3 angular brackets. The grid and blocks are characterized by 1, 2 or 3 dimensions, x, y and z respectively. The choice of these dimensions is usually made according with the data structures to handle. The architecture of a CUDA grid is reported in fig. 3.7 and the coordinates order is (x, y, z).

In order to identify any thread instantiated into the grid, we can use the following variables:

- **blockDim**: it comes with 3 fields (x, y, z) in which the block dimensions are saved;
- **blockIdx**: this is a unique identifier of a block within the grid. The blocks are labelled with progressive numbers starting from 0 to $n - 1$, where n is the number of initialized blocks;
- **threadIdx**: this is a unique identifier of a thread within a block. The threads are labelled with progressive numbers starting from 0 to $n - 1$, where n is the number of initialized threads.

¹dim3 is a built-in variables which specifies the grid and block dimensions.. They are only valid within functions that are executed on the device.

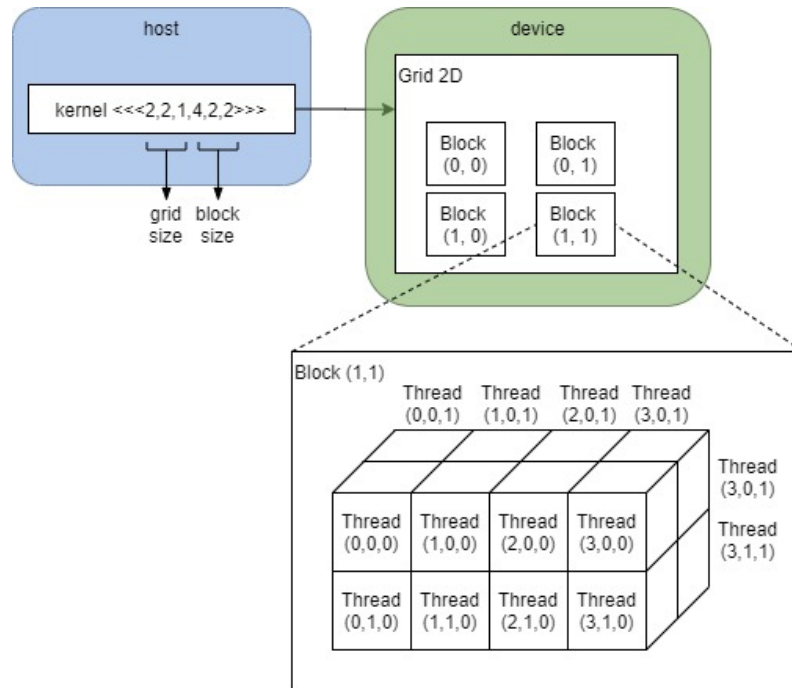


Figure 3.7: Multidimensional CUDA grid

By virtue of the use of these variables, a thread can be identified as stated in listing 3.2.

```

1 // gridSize_BlockSize
2
3
4 __device__
5 int getGlobalIdx_1D_1D() {
6     return blockIdx.x * blockDim.x + threadIdx.x;
7 }
8 __device__
9 int getGlobalIdx_1D_2D() {
10    return blockIdx.x * blockDim.x * blockDim.y
11        + threadIdx.y * blockDim.x + threadIdx.x;
12 }
13 __device__
14 int getGlobalIdx_1D_3D() {
15    return blockIdx.x * blockDim.x * blockDim.y * blockDim.z
16        + threadIdx.z * blockDim.y * blockDim.x
17        + threadIdx.y * blockDim.x + threadIdx.x;
18 }
19
20
21 __device__
22 int getGlobalIdx_2D_1D() {
23     int blockId = blockIdx.x + blockIdx.y * gridDim.x;
24     int threadId = blockId * blockDim.x + threadIdx.x;
25     return threadId;
26 }
27 __device__
28 int getGlobalIdx_2D_2D() {

```

```

29     int blockIdx = blockIdx.x + blockIdx.y * gridDim.x;
30     int threadIdx = blockIdx * (blockDim.x * blockDim.y)
31                   + (threadIdx.y * blockDim.x) + threadIdx.x;
32     return threadIdx;
33 }
34 __device__
35 int getGlobalIdx_2D_3D() {
36     int blockIdx = blockIdx.x + blockIdx.y * gridDim.x;
37     int threadIdx = blockIdx * (blockDim.x * blockDim.y * blockDim.z)
38                   + (threadIdx.z * (blockDim.x * blockDim.y))
39                   + (threadIdx.y * blockDim.x) + threadIdx.x;
40     return threadIdx;
41 }
42
43
44 __device__
45 int getGlobalIdx_3D_1D() {
46     int blockIdx = blockIdx.x + blockIdx.y * gridDim.x
47                   + gridDim.x * gridDim.y * blockIdx.z;
48     int threadIdx = blockIdx * blockDim.x + threadIdx.x;
49     return threadIdx;
50 }
51 __device__
52 int getGlobalIdx_3D_2D() {
53     int blockIdx = blockIdx.x + blockIdx.y * gridDim.x
54                   + gridDim.x * gridDim.y * blockIdx.z;
55     int threadIdx = blockIdx * (blockDim.x * blockDim.y)
56                   + (threadIdx.y * blockDim.x) + threadIdx.x;
57     return threadIdx;
58 }
59 __device__
60 int getGlobalIdx_3D_3D() {
61     int blockIdx = blockIdx.x + blockIdx.y * gridDim.x
62                   + gridDim.x * gridDim.y * blockIdx.z;
63     int threadIdx = blockIdx * (blockDim.x * blockDim.y * blockDim.z)
64                   + (threadIdx.z * (blockDim.x * blockDim.y))
65                   + (threadIdx.y * blockDim.x) + threadIdx.x;
66     return threadIdx;
67 }

```

Listing 3.2: CUDA threads indexing [11]

Concretely, on the hardware, the CUDA architecture is built around a scalable array of multithreaded Streaming Multiprocessors (SMs) as shown in fig. 3.8. Each SM has a set of execution units, a set of registers and a chunk of shared memory. Each SM can execute concurrently a fixed number of threads grouped into multiple structures called *wraps*.

Another interesting programming language, within the field of GPGPU programming, is the one provided by the framework OpenCL. This language, on par with CUDA, allows to extend the syntax of certain languages through dedicated run-time API. In this way the programmer is able to manage the parallelism and data exchange in massively parallel applications. OpenCL is a standardized programming model by which it is possible to implement parallel cross platform applications, that can run on several kind of GPU (NVIDIA, AMD, etc) without changing the code. Since CUDA

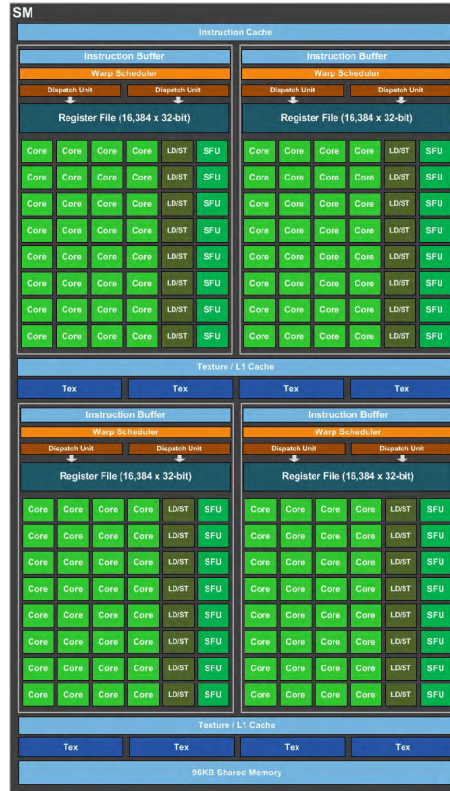


Figure 3.8: NVIDIA GPU GeForce Streaming multiprocessor structure

is a proprietary code, self-made by NVIDIA, it generally offers better performances on GPU CUDA-capable with native language, because of the direct support the company can assure. A concrete example of restrictions on OpenCL system run on GPU NVIDIA, is the DECINE-X model:

- if OpenCL is enabled, only 1 GPU can be used;
- if CUDA is enabled, 2 GPUs can be used for GPGPU.

Chapter 4

QoG

This chapter explains the main features of the QoG system that has been developed, in order to accelerate large data modeling for quantum computation. The chapter is subdivided into six parts: introduction, requirements definition, technological choices, design, implementation, testing and future developments.

The purpose of all of this is to realise and develop a system capable of creating a QUBO model in the fastest and smartest way, on the basis of a specific problem as defined by the user.

As already stated in the section 2.2.2, QUBO model can represent a given binary quadratic optimisation problem by describing it through an upper triangular matrix.

Using a traditional approach, this model require very long time to be solved, but thanks to the brand new approach of quantum annealing, a possible optimal solution can be found in shorter time. However, getting a real speedup, also requires to accelerate the creation of the QUBO matrix: this particular aspect is a sensitive real bottleneck.

The modeling and technological problem of creating these matrices is an old issue that the quantum annealing has recently unearthed, but we do not have efficient techniques for this purpose at the present time. The core of this project is to create an efficient memory storage of this model, making use of the parallel computation on GPU; in this way, we can obtain a general abstract system, valid in a wide range of cases.

The system efficiency has to be verified by evaluating two different aspects:

- **time**: given the large dimensions of the matrices minimising the time requires the use of the proper technologies;
- **space**: given the large dimensions of the matrices minimising the space requires the use of the proper storage techniques;

4.1 Requirements analysis

The system should be able to create an appropriate QUBO matrix for each given problem to solve. If the particular problem being analyzed has certain constraints to comply with, they have to be handled by modifying the appropriate costs, namely the entries of the QUBO matrix. The matrix has to be generated starting with the

user defined unconstrained costs. Those costs can be altered by properly adding or subtracting the value related to each constraint. Once the QUBO matrix is defined and generated, it is necessary to save it permanently on the file system, so that the matrix can be used by other systems to evaluate its solutions. For this system the main spotted use case are as follows:

- user problem definition (`define_problem`);
- constraints definition (`add_constraint`);
- building of the QUBO matrix (`build_QUBO`);
- matrix saving (`save_matrix`).

4.1.1 Problem definition

The problem has to be described in a very easy and compact way. The most common problem to be solved with this kind of approach are of the RAP type, that is resource allocation problems. In this kind of problem, the main goal is to find the best assignment for a certain job set, starting with a given available resource set. Defining a problem by using a QUBO matrix requires, firstly, the identification of the proper number or variables to be used to set the matrix dimension. The number of variables, for RAP problems, can be calculated as the product of the number of available resources and the number of jobs to be performed, as every variable represents the association resource-job. Some examples of these kinds of problems are shown in table 4.1.

resources	jobs
trucks	routes
teams	projects
workstations	employees
radio antennas	radio frequencies

Table 4.1: Resource allocation problem examples

Once the number of variables has been defined, specifying the associated costs to the different combinations of variables for filling the QUBO matrix is needed. It must be possible to define such costs either in a customized way or using pre-built templates: Uniform, Ascending, Descending, Random. Examples of these pre-built templates, as applied to a 3×3 QUBO matrix, are shown in fig. 4.1.

Seeing more in detail some aspects about the definition of the customized costs: it all starts with an homogeneous 0 cost for every entries of the matrix, then the cost related to a certain i, j combination must be assigned in an independent manner from the others. An example of a QUBO matrix obtained by the custom problem.csv file is shown in fig. 4.2 (the file is given in the left half of the image).

Every information about a single problem must be defined in a specific configuration file, which can be of two different types:

0	1	2
0	3	4
0	0	5

a. type_costs: Ascending
base_costs: 0

5	4	3
0	2	1
0	0	0

b. type_costs: Descending
base_costs: 5

1	1	1
0	1	1
0	0	1

c. type_costs: Uniform
base_costs: 1

2	5	1
0	3	4
0	0	5

d. type_costs: Random
base_costs: 5

Figure 4.1: QUBO costs examples

resource_number: 2 jobs_number: 2	R2J2	00	01	10	11
	R1J1				
R1	J1	R2	J2	Cost	
0	0	0	0	3	
0	0	0	1	2	
0	1	1	1	8	
1	0	1	0	4	
1	1	1	1	10	

00	3	2	0	0
01	0	0	0	8
10	0	0	4	0
11	0	0	0	10

Figure 4.2: QUBO costs loaded from problem.csv file

- **problem.conf**: this is a concise and synthetic file and for that reason highly recommended for problems with a large number of variables. The problem can be fully defined through the use of only 4 fields:
 - **resources number**: to define the number of resources;
 - **jobs number**: to define the number of jobs;
 - **type cost**: to define the costs' type. The possible values for this field are: Uniform, Ascending, Descending and Random;
 - **base cost**: the primary cost from which all the other costs are derived. For a Uniform value, every cost is to be set as *base cost*, for Ascending and Descending values, the first cost is the *base cost* (the others being set to the result of the sum of ± 1) whilst for a Random value, it will be the upper bound of random numbers.

- `problem.csv`: this file is more detailed and flexible than the previous one, in order to define the resource number, the job number and the cost of every i, j index combination.

4.1.2 Constraints definition

Once that the primary costs of the QUBO matrix are defined, any further constraint is to be imposed. These constraints are specific rules, stemmed from the particular semantic of the correlation between two variables. Given all the possible resource-job combinations, a constraint can be applied on a subset of this set (i.e. all the combinations of variables with the same resource value). Once a constraint is fully defined, the pairing variables can either respect it or not, or even not being influenced by it. All these scenarios imply the potential change of the cost value, in the following manners:

- if an i, j couple is not influenced by the constraint, then its cost does not change;
- if an i, j couple is influenced by the constraint and it obeys, then its cost changes.

More to the point and by virtue of the quantum annealing properties, there is an inverse relationship between the cost value of a certain i, j combination and the probability it will be part of the final solution: namely, the more the cost value increases, the more the probability will decrease and vice versa, so in order to promote a particular combination, we need to decrease the related cost. The value to be added or to be subtracted directly depends on the importance of the constraint: the more this value is high, the more the importance will rise. A particular attention has to be given to this value because it could lead towards infeasible solutions.

A simple and trivial example can be obtained by assigning one job per resource. In this case it will be necessary to decrease the cost of every association marked by the same resource and job. Meanwhile to the associations with the same resource but different job will "give a penalty" by increasing the cost. In the end the associations with different resources in their variables will not be altered on this as not subjected to the specific constraint. In this case the associations with the same resource-job lie on the main diagonal of the QUBO matrix. In fig. 4.3 a constraint applied to a matrix with all 0 entries with 1 resource ($R0$) and 3 jobs ($J0, J1, J2$) is shown.

4.1.3 Building of the QUBO matrix

At this point, once all the necessary data and information - costs and constraints - are known, the system needs to be able to give an internal representation of the QUBO matrix. By the term internal representation we intend the saving of the matrix in an appropriate data structure into the system. Two non-functional requirements come into play by now:

- **time**: the matrix creation must take place as quickly as possible, trying to maintain a constant time elapsed in the face of a suitable number of additional variables and, therefore, a remarkable matrix size increase;

	R0 J0	R0 J1	R0 J2
R0 J0	0	1	2
R0 J1	0	3	4
R0 J2	0	0	5

Initial QUBO matrix

	R0 J0	R0 J1	R0 J2
R0 J0	-1	2	2
R0 J1	0	-1	2
R0 J2	0	0	-1

Constraint matrix

	R0 J0	R0 J1	R0 J2
R0 J0	-1	3	4
R0 J1	0	2	6
R0 J2	0	0	4

Final QUBO matrix

Figure 4.3: QUBO constrained example

- space: creating a matrix shall not take up too much space but as little space as possible, in view of optimizing the chosen data structures.

4.1.4 Matrix saving

The last step required is now that of saving the matrix to disk, so it can be associated to different kind of linear systems and so reaching optimal solutions to the primary problem. The storage can be performed in two different ways:

- whole: it must be possible to save the whole matrix on a .csv or .txt file. By the term whole matrix we intend the matrix with all its entries, even those under the main diagonal (which are all set to 0, being this matrix an upper triangular one). By doing that, it is possible to speed up the allocation of the data structure related to other systems aimed to evaluating the solution;
- compact: if needed, only the not null portion of the matrix ought to be saved on a .csv or .txt file. In this way there is an optimization as regards the storage capacity consumed on the disk.

4.2 Design

The system is mainly composed by 4 parts to be run one by one:

1. uploading the problem data;
2. defining the constraints;
3. creating the matrix;
4. saving the data.

All the users have to do, is defining the problem data and the constraints to impose, so the application can create and save the specific matrix in the desired format (.csv or .txt). As it will be shown in the remainder of the discussion (specifically in the section 4.3), a general-purpose computing on graphics processing units programming

approach was chosen in order to take full advantage of the parallelizable properties of creating a matrix. More specifically, using CUDA as programming language, the constraints to be applied can be directly implemented in an ad-hoc kernel: in every single kernel, more than one constraint can be placed into it, so only one kernel with any mandatory constraint is necessary. Every information concerning the problem, as user-defined in the initial configuration file, is loaded throughout a specific component called `DataLoader`. This last one is responsible for the wrapping in the information within an object called `ProblemInfo`, which keeps a record of any fundamental information about the problem under consideration. The constraints to be applied to the matrix are implemented in a CUDA kernel. Once every necessary information is defined, the `QUBOBuilder` component deals with both the matrix creation on the GPU and the model saving on the file system. fig. 4.4 gives the main stream of the system.

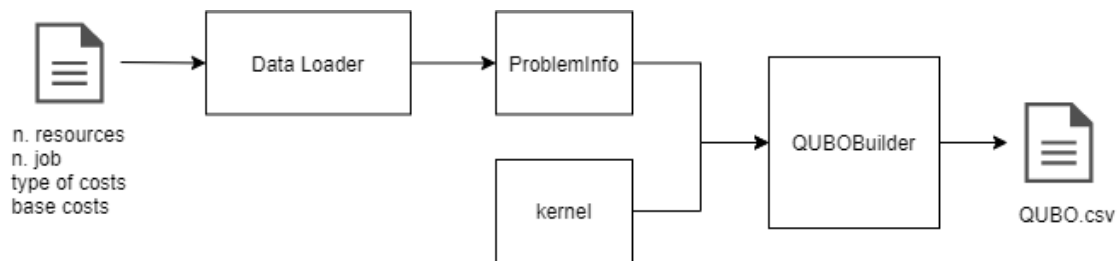


Figure 4.4: System main data flow

fig. 4.5 shows the main system architecture as emerged from the requirements analysis and a preliminary draft. In this UML diagram, we can see the main classes of the model, the latter being responsible for:

- data loading;
- kernel and constraints definition;
- creating and saving the QUBO matrix.

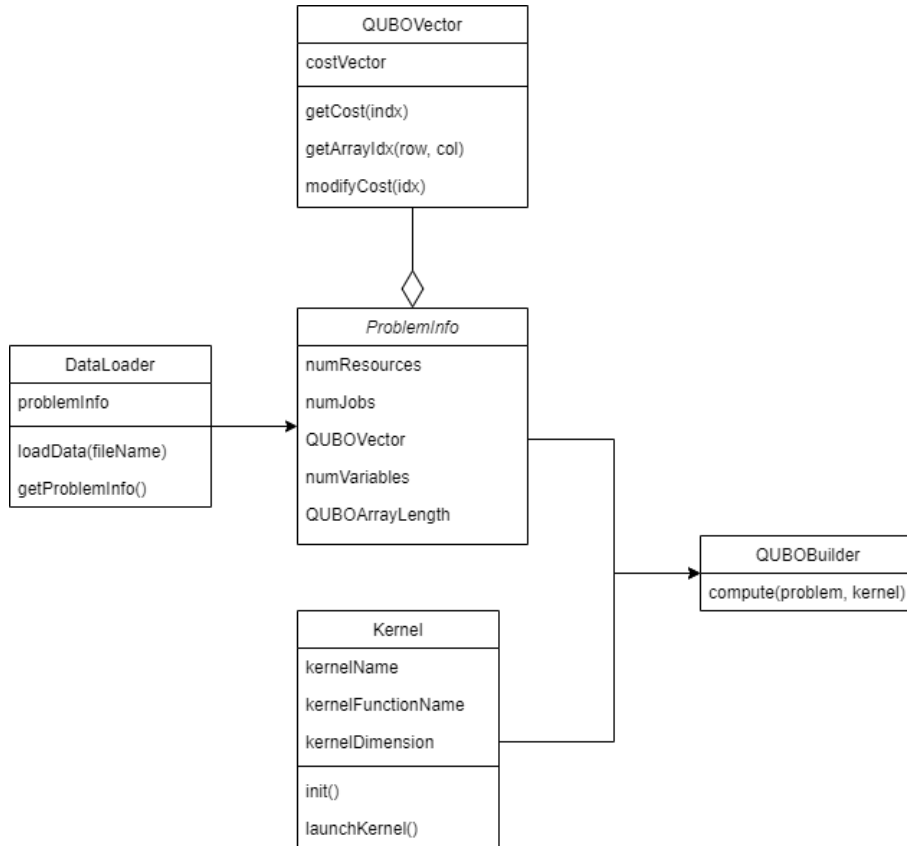


Figure 4.5: System architecture

4.2.1 Uploading the problem data

This part of the system is mandated to upload the data contained in the configuration file, to properly parse and store them into an `ProblemInfo` object. Since a configuration file contains different information, everyone must necessary be parsed in different ways, however, in both cases information are to be put in a `ProblemInfo` object. These information are:

- the number of resources;
- the number of jobs;
- the number of variables;
- the length of the QUBO array;
- the vector with the unconstrained costs.

According to the requirements emerged during the analysis phase, the problem data can be defined through the following files:

- `problem.conf`, more synthetic;
- `problem.csv`, more flexible.

In both cases the number of resources and jobs is automatically taken from the text while the number of variables is calculated as their product. The real difference stands in the generation of the costs vectors: in this respect it is recalled that the vector is a row-major order representation of the upper triangular part and the main diagonal of the QUBO matrix. The length of the array is given by $\frac{N*(N+1)}{2}$ where N is the number of variables. If the data are uploaded from the file `problem.conf`, the array will be created following the rules spotted by the parameters `type_cost` and `base_cost`; if they are taken via the file `problem.info`, it will be necessary to create an array of zeros for the costs that will be arranged on the basis of the configuration defined in the file. In fig. 4.6 a graphic representation of the above is given.

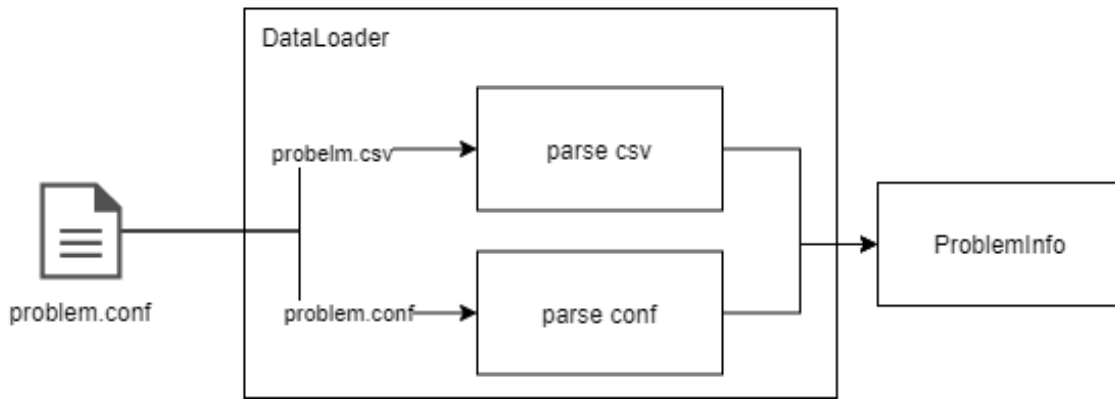


Figure 4.6: DataLoader flow

4.2.2 Defining the constraints

The constraints for the QUBO matrix depend on the problem semantics, in particular on the properties elapsing in any i, j resource-job pair. Depending on the will of enhancing or not a certain i, j couple in the final solution, every cost in a certain i, j position in the matrix may decrease or increase. In order to define a constraint, given a variable it is therefore essential being able to trace its specific resource-job combination R, J back. When all these information are traced back, both for variables i and j , “shaping” the constraint is now made possible. For every i, j cell, if that association is affected by the constraint and the constraint is respected, the cost will decrease of an α factor. Otherwise the cost will rise of a β factor. Coefficients α and β may have the same value and are user defined. As described in the Technologies section, the constraints to apply to the matrix will be implemented in an ad-hoc CUDA kernel. This phase is represented in fig. 4.7.

4.2.3 Creating the matrix

The step of creating the matrix is the very crux of the whole system and therefore its performances. At this point we need to upload any information about the problem itself in a `ProblemInfo` object and set the kernel we are going to use to apply the constraints. Thereafter we can move on to the actual creation of a brand

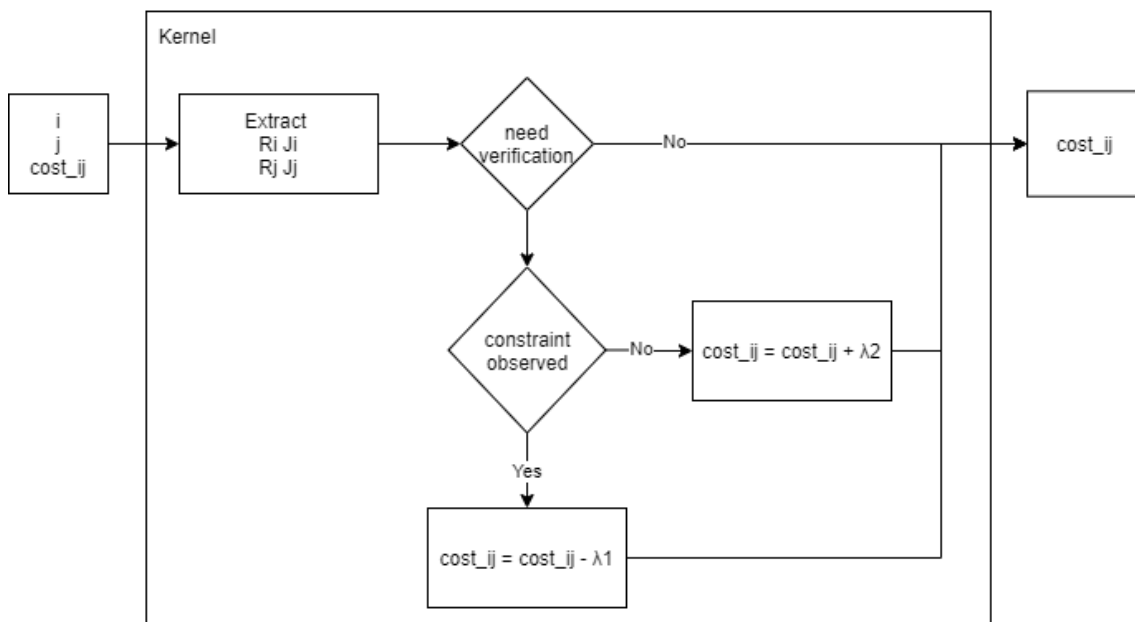


Figure 4.7: Single constraint definition

new QUBO matrix using the function `compute(problem, kernel)` provided by the QUBOBuilder object. The inputs of this particular function are all the information needed for the creation and, as a result of its invocation, it will copy every data structure from the CPU to the GPU and finally recall the kernel. The new matrix will be allocated on the GPU variable `d_costVector` so that copying this structure on the CPU will allow to de-allocate the occupied space on the GPU, as shown in fig. 4.8.

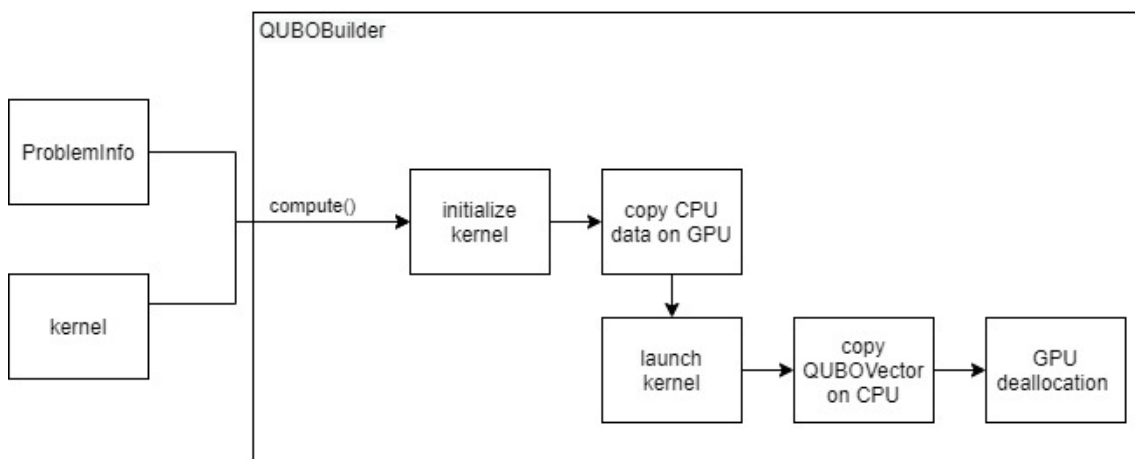


Figure 4.8: QUBO matrix computation

Due to the particular nature of the matrix, we will need to save only the upper triangular portion of the matrix - entries above the main diagonal and the diagonal itself - in order to optimize the space occupied by the data structures, both on the CPU and the GPU. As seen before, a row-major order model has been chosen so

that the matrix can be represented through a one-dimensional array. This model provides for the saving a single matrix placing each row side-by-side and saving about to 50% the space taken up. In fact, if we notice that the rows of the upper triangular section decrease linearly, then combining them side-by-side we obtain a $\frac{N*(N+1)}{2}$ length array, where N is the number of variables. fig. 4.9 provides a simple but exhaustive example of what has just been stated.

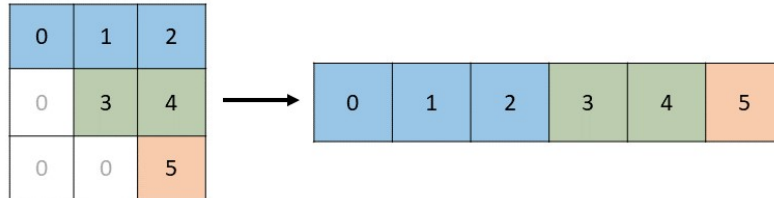


Figure 4.9: Row-major model

So we just presented some important issues in terms of space reduction. Regarding instead the computing time, it is possible to say that this time can be optimized too, by virtue of the parallel computation on GPU. On the assumption that the value of any cell is independent of the other, the constrained cost of each cell of the matrix can be evaluated in parallel. Since we register a diminution, in terms of time, from $\Theta\left(\frac{N*(N+1)}{2}\right)$ to $\Theta(1)$, we will have a total time reduction of a $\mathcal{O}(N^2)$ factor. Technically, the computing time on the GPU will not be exactly $\Theta(1)$, because of the costs associated to the allocation of thread-blocks grid that CUDA uses to compute the kernel. But, despite this, there is a substantial reduction of the operational timelines.

Any kernel vowed to the evaluation of the final matrix is totally identified through the following parameters:

- `kernelName`: the name of the .cu file containing the kernel;
- `kernelFunctionName`: the name of the function implemented within the file to be run on the GPU;
- `kernelDimension`: the dimension of the CUDA grid to be used for the computation of the kernel.

The available functions for a kernel object are:

- `init()`: the function used to initialize the base model of JCuda driver;
- `launchKernel()`: the function used to launch the kernel.

So it will be possible to define a different kernel for every problem to solve, we opted for a pattern factory [26] in order to facilitate both the new kernel definition and its usage. The UML scheme is shown in fig. 4.10. The dimensions of the CUDA grid which the kernel can be launched with are wrapped in a specific object called `kernelDimension`. The pattern factory has been used again for the grid dimension, given that different sizes are available for that component. In fig. 4.11 there is the UML model for that.

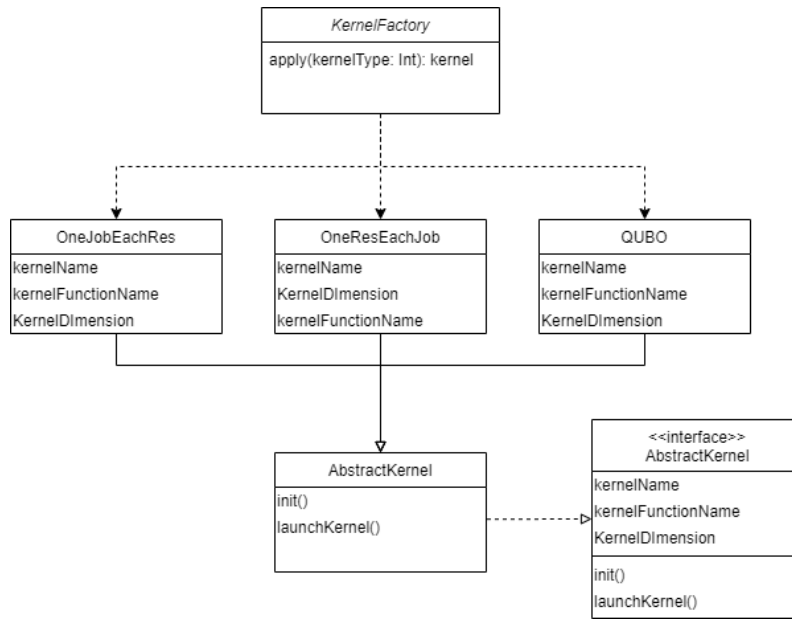


Figure 4.10: Kernel Factory

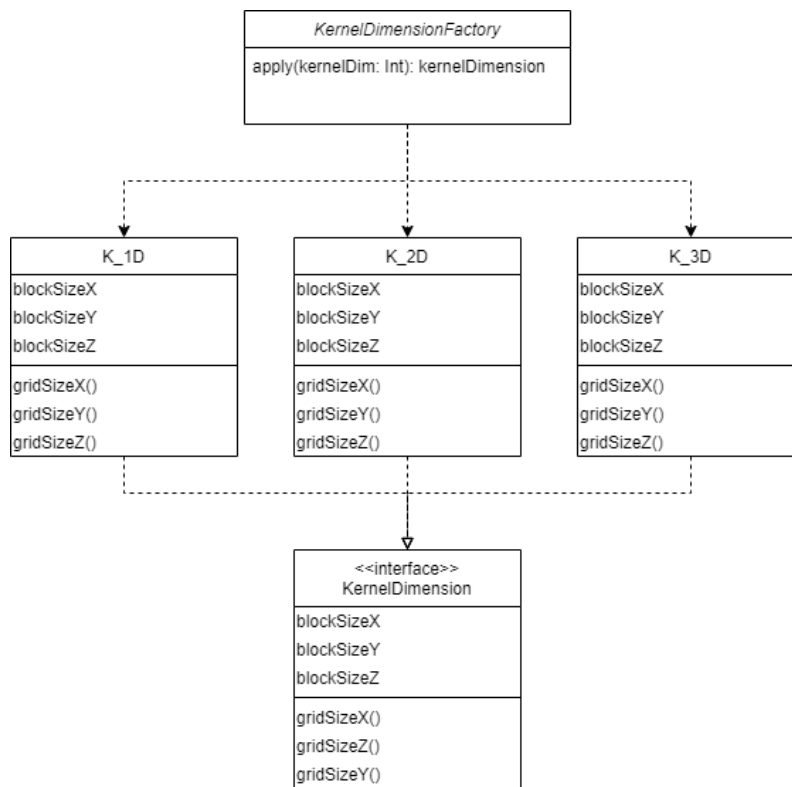


Figure 4.11: KernelDimension Factory

4.2.4 Saving the data

Two different ways of saving the matrix must be handled:

1. **whole**: in this case the matrix has to be saved $N \times N$, so even the portion of zeros under the main diagonal. To do so, it will be required to add a certain

number of 0 to the left of the row equivalent to the row index, i.e. no adding for the first row ($i = 0$), one 0 for row number 1 ($i = 1$) and so on until the reaching of $i = N - 1$. This operation inputs are QUBOVector with the values to be stored, the number N of variables and the name of the file on which to save the data;

2. **compact**: in this manner it is necessary to save the costs array as reported on the file only. Inputs are the QUBOVector with the values to be stored and the name of the file on which to save the data.

A UML representation of the saving process is given in fig. 4.12.

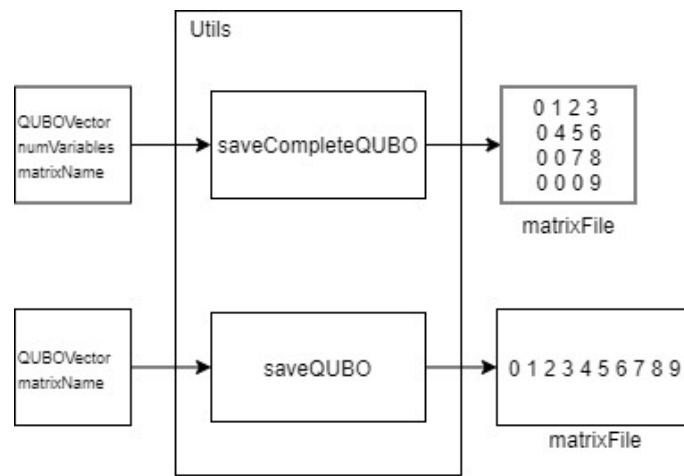


Figure 4.12: Matrix saving functions

4.3 Technologies

Choosing the appropriate technologies is a fundamental step for the success of the work and so two aspects have been deeply analysed:

1. the main architecture of the project requires the use of an appropriate programming language such as Scala. This particular choice is due to the fact that Scala is simultaneously an object and functional language, in addition it has advanced mechanisms, such as type members and implicits;
2. time optimisation and the parallelisation of the system are the main goals of the project and also the reason why CUDA has been adopted. In fact CUDA, among all the existing GPGPU frameworks (i.e. OpenCL, Vulkan, etc), makes it possible to take full advantage of the computational power of the computers available (equipped with NVIDIA GeForce GTX GPUs).

4.3.1 Scala

The main structure of the application has been conceived using Scala as programming language. Scala allows to create really complex systems which can be also extended through a very concise and readable code: the combination of the object oriented and functional paradigms provides a unique and highly versatile language.

As just mentioned in the opening of the chapter, we used two main advanced Scala functionalities:

- type members;
- implicits.

Type members

Type members are abstract terms defined within a trait, a class or an object. These terms, identified via the keyword “type”, can be used as alias for other default types (i.e. `Int`, `Array[Double]`, etc). Once a new type member `type T = String` is defined, it will be automatically turned into a `String` type during the compilation, and this will happen every time a `T` type is used; for this reason any type member can also be treated as a built-in type in order to define variables and the parameters of the functions.

An example of using type members to easily create simple alias for more complicated data types is reported in listing 4.1:

```
1 class Book {  
2     import scala.collection.mutable._  
3     type Index = HashMap[String, (Int, Int)]  
4 }
```

Listing 4.1: Type members example [37]

In this particular case, the alias *Index* enables to write a better code in terms of linearity, conciseness and expressiveness, and especially using the same concrete-type semantic. In the end the code will be more readable and so easier to maintain.

Another important feature of type members is that of being capable to establish Abstract types in trait or abstract class, which in turn are concretely defined in inherited concrete classes. We will not discuss this any further, because this functionality has not been used for our purposes.

Implicits

Implicits are normal unary functions marked with the *implicit* keyword from the beginning. The compiler uses these functions to automatically convert the objects type if needed without to explicitly recall the method in the code. The implicit are used every time the type checker finds a type-error in the code and, in this case, it will verify if a proper conversion exists in the scope on its own, so the error can be fixed. Some rules the implicits are to follow are [37]:

- marking rule: a function, to be an implicit, must be marked with the keyword *implicit* at the beginning of its signature;
- scope rule: an implicit has to be present in the code scope in order to be applied;
- non-ambiguity: implicit conversion are used by the compiler to fix typing problem as a last resort;
- one-at-a-time rule: every conversion may provide for one implicit at most;
- explicit-first rule: whenever code type checks as it is written, no implicits are attempted.

The implicits can be used in three different cases:

1. the expression type is different from the expected one;
2. an object member to which you are trying to accede does not exist;
3. the actual parameters are different from the expected when recalling a function.

An example of definition and use of an implicit is given in the listing 4.2:

```

1
2 case class Complex(re: Double, im: Double) {
3     def +(that: Complex) = Complex((this.re + that.re), (this.im +
4         that.im))
5 }
6 def printComplex(c: Complex) = println(s"real: $c.re    imaginary: $c.im
7     ")
8 implicit def IntToComplex(x: Int): Complex = Complex(x, 0)
9

```

```

10 x: Int = 1
11 z: Complex = Complex(1, 2)
12
13 z + x //1. Complex class does not have +(Int)
14
15 x.re //2. Int class does not have re member
16
17 printComplex(x) //3. the printComplex function does not accept Int
    parameter

```

Listing 4.2: Implicits example [30]

The implicits also apply to define some function parameters. If a function parameter is marked with the keyword *implicit* and it is not given when the function is called, Scala compiler will provide for that assignment. Since we did not use this functionality, we only mention it exists.

4.3.2 CUDA and JCuda

The main reason why CUDA has been chosen for the parallel creation of the QUBO matrix is because it allows to manage heterogeneous systems in which CPU and GPU computation coexist. The particular notion of `kernel` is the way CUDA permits us to handle the GPU elaboration. Throughout this kind of abstraction, the SPMD paradigm ensures the definition of the operations set to be performed on a large amount of data. In addition, synchronisation techniques or multiprocessor streaming have not been called into action, because any constraint is run by a single kernel.

However, CUDA and Scala are not directly inter-operable: in fact NVCC - the CUDA compiler - cannot be used to compile Java code and, vice versa, Java does not support any CUDA extension. For example the CUDA call `kernel<<<gridSize, blockSize>>>` cannot be used in Java. To circumvent this, we used the JCuda framework [10] to run CUDA code from Java and therefore from Scala. JCuda offers a large set of Java classes that, by using the Java Native Interface (JNI) and the CUDA Driver API, makes possible to employing any CUDA function directly from Java. JCuda enables Java programs to call CUDA kernels directly from Scala CPU-Code. In order to call the CUDA kernels the GPU-Code, including kernel and device functions, have still to be written in C, and should be converted into PTX or CUBIN file by NVCC compiler. There are two ways to compile GPU-Code [34] both represented in fig. 4.13:

- **Just-In-Time (JIT) Compilation:** Java programs call a Java API, `Runtime.getRuntime()` to let NVCC compile GPU-Code into PTX or CUBIN;
- **Off-line Compilation:** GPU-Code is compiled in advance. Finally, JCuda programs can load kernels from PTX and CUBIN files through JCuda Driver API, `cuModuleLoad`, for the execution on GPU.

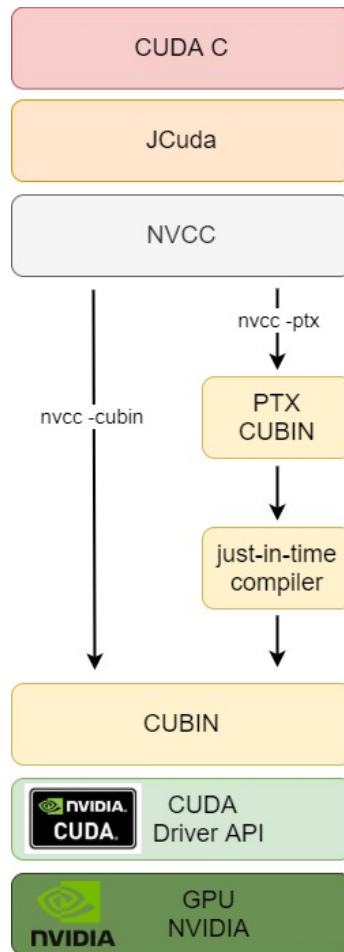


Figure 4.13: NVCC CUDA compilation

Precompiled kernels, both in *PTX* and *CUBIN* format, can be run thanks to API CUDA Drivers [41]. Now, most of the time, kernel run by the implemented system will be compiled into *CUBIN* format, in order to avoid to recall the just-in-time CUDA compiler for any implementation, and to translate *PTX* files in *CUBIN* code that can be executed on the specific GPU architecture. In the JCuda samples repository [33] we can find the methods provided to run the just-in-time compiler.

The useful functions JCuda offers to interact with the GPU are:

- **Kernel compilation** through a function to recall directly the NVCC compiler;
- **GPU'S memory allocation and de-allocation** using the dedicated function `cuMemAlloc()` and `cuMemFree()`;
- **CPU to GPU data transfer** through the functions `cuMemcpyHtoD()` and `cuMemcpyDtoH()`;
- **loading of the file containing the kernel** using `cuModuleLoad()`;
- **kernel function loading** through the function `cuModuleGetFunction()`;

- **definition of the parameters as input for the kernel** by a Pointer abstraction: this class identifies a pointer in the same way C language does, but as referred to the GPU memory;
- **kernel execution** by using `cuLaunchKernel()`;
- **synchronisation between CPU and GPU** using the function `cuCtxSynchronize()`.

As a consequence of the technological choices we made, the system architecture can be given as drawn in fig. 4.14.

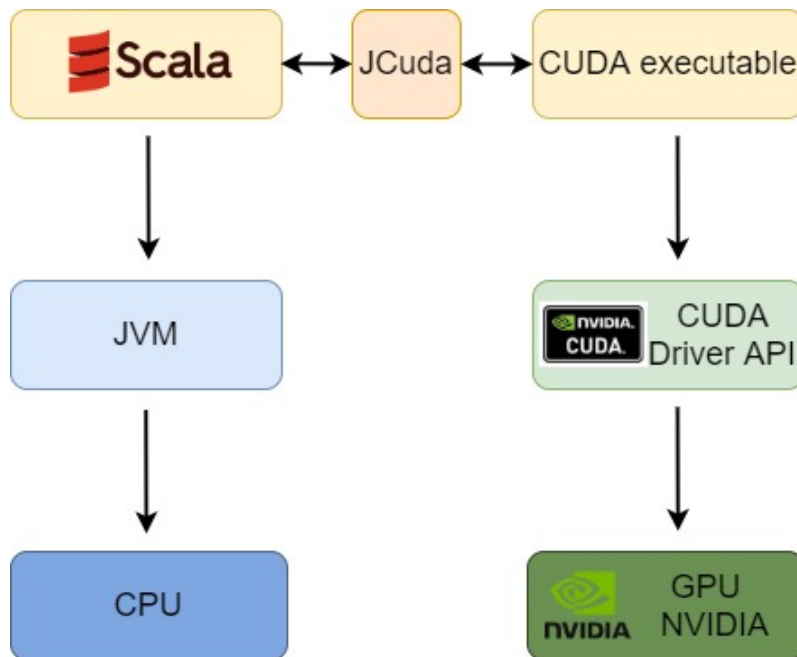


Figure 4.14: Technological stack

4.4 Implementation

As already mentioned in the Technologies section, the project has been realised using two different programming languages:

- Scala: for the creation of the system infrastructure;
- CUDA: for the computing time optimisation in creating the final QUBO matrix.

4.4.1 Scala advanced features

During the analysis phase, it has been noticed that the matrix can be composed of both Int and Double numbers. Distinguishing these two kinds of data is very important, especially for the GPU allocation stage. Moreover, in case of Int numbers, it is possible to save half of the occupied space in memory: in fact, it takes only 4 bytes to save an Int, while a Double requires exactly twice (8 bytes). The new Scala type member `NumType` identifies the nature of the values to put in the QUBO matrix, interchanging between Int and Double. This idea has found application also for the `Cost` type, which associates the name of a cost type and the `NumType` array with the actual costs. So, by using the type members technique we can define the more appropriate data type for any problem to solve, without having to edit the code once more. The code snippet in listing 4.3 is related to the definition of these new data types.

```

1 object Cost {
2     type NumType = Double
3     type Cost = (String, Array[NumType])
4 }
5 
```

Listing 4.3: Type members definition

Another interesting issue that Scala offers, which has proven itself quite resourceful, is the implicit that has been already introduced before. It provides automatic conversions managed by the compiler whenever necessary. This worthwhile functionality has been very useful for this project particularly as regards the conversion from `Array[Int]` to `Array[Double]` and vice versa. The code in listing 4.4 is related to the creation of these implicits.

```

1
2 implicit def intToDouble(a: Int): Double = a.toDouble
3
4 implicit def ArrayIntToArrayDouble(a: Array[Int]): Array[Double] = a.
    map(_ . toDouble)
5
6 implicit def ArrayDoubleToArrayInt(a: Array[Double]): Array[Int] = a.
    map(_ . toInt)

```

Listing 4.4: Implicits definition

4.4.2 Indexing

Another important implementation issue is worth underling regards the QUBO matrix creation: as mentioned before, the matrix is represented as a $\frac{N*(N+1)}{2}$ length one-dimensional array using the row-major model, where N is the variables number. Once the unconstrained cost matrix is mapped to the related array, the new constrained costs are evaluated in parallel running a CUDA kernel. More specifically, each CUDA thread will evaluate a single cell of the costs array and the index of the cell to be evaluated is given by the CUDA thread id: for example, a thread with id=0 will compute the value related to the first cell of the QUBO array, a thread with id=1 the one to the second cell and so on. However, as described in the Design section, knowing the pair resource-job related to an i, j couple is needed if we want to apply any constraint to that particular i, j . So, to track these information back, we need, first of all, to define the row i and column j index of a certain cell starting from the linear index. Despite the complexity increases in the kernel managing, what has been presented allows to avoid any kind of cycle within CUDA threads. The process of switching from the linear index of the matrix to the resource-job definition for a generic i, j couple is given in fig. 4.15.

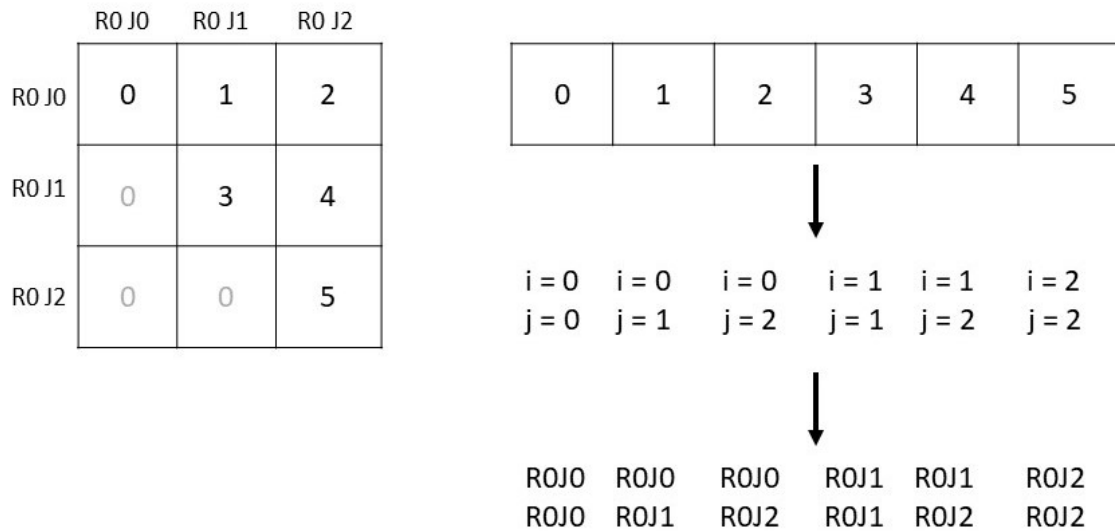


Figure 4.15: Resources and Job computation

To trace back the row-column pair of a certain linear index of an upper triangular matrix, for which the row-major model has been used, we inverted the formulas we used earlier to find the linear index, starting from rows and columns of the matrix itself.

Given the parameters:

- n the number of variables;
- k the linear index;
- i the row index;

- j the column index.

We can write as follows:

$$k = j + ni - \frac{i(i+1)}{2} \quad (4.1)$$

$$j = k - ni + \frac{i(i+1)}{2} \quad (4.2)$$

$$i = \lfloor \frac{2n+1 - \sqrt{(2n+1)^2 - 8k}}{2} \rfloor \quad (4.3)$$

We assume that all the data structures indices (QUBO array, resources and jobs) start with 0 and are linearly increasing of one unit each. In this way, once the row and column indices are obtained, the resource-job ids can be calculated as follows:

$$\begin{aligned} R_i &= \lfloor \frac{i}{z} \rfloor & R_j &= \lfloor \frac{j}{z} \rfloor \\ J_i &= i \bmod z & J_j &= j \bmod z \end{aligned}$$

Listing 4.5 presents the code used within the CUDA kernel to get the resource-job ids as referred to a certain cell (k) of the QUBO array. The linear index is determined following the indexing operation of the one-dimensional CUDA grid threads, as stated in the 3.2 section.

```

1
2 int k = blockIdx.x * blockDim.x + threadIdx.x;
3
4 int i = floor((-sqrt((2*n+1) * (2*n+1) - 8*k) + 2*n+1) / 2);
5 int j = k + n*i + i*(i+1) / 2;
6
7 int res_i = floor(i / z);
8 int job_i = i % z;
9
10 int res_j = floor(j / z);
11 int job_j = j % z;
```

Listing 4.5: resources and jobs ids computing

Listing 4.6 shows the CUDA kernel developed to build the matrices used during the testing phase.

```

1
2 extern "C"
3 __global__ void build_matrix(int num_elem, int num_var, int num_job,
4 double *mat) {
5     int tid = blockIdx.x * blockDim.x + threadIdx.x;
6     if(tid < num_elem) {
7
```

```

8      int row = floorf((-sqrtf((2*num_var+1)*(2*num_var+1)-8*tid)+2*
9          num_var+1)/2);
10     int col = tid + row - row*(2*num_var-row+1)/2;
11
12     int res1 = floorf(row / num_job);
13     int job1 = row % num_job;
14
15     int res2 = floorf(col / num_job);
16     int job2 = col % num_job;
17
18     int lambda1 = 1;
19     int lambda2 = 1;
20
21     //Constraint 1
22     if(res1 == res2 && job1 == job2) {
23         mat[tid] = -1*lambda1 + mat[tid];
24     } else if (res1 == res2 && job1 != job2) {
25         mat[tid] = 2*lambda1 + mat[tid];
26     }
27
28     //Constraint 2
29     if(res1!=res2 && job1==job2) {
30         mat[tid] = 1*lambda2 + mat[tid];
31     }
32 }
33 }

```

Listing 4.6: CUDA kernel implementation

This kernel is launched within the function `Kernel.launchKernel()` called into the method `QUBOBuilder.compute()`. The launching of the kernel is performed by using the API provided by JCuda: in particular, to launch the execution of a kernel we used the function listed in listing 4.7.

```

1  cuLaunchKernel(function ,
2      gridX, gridY, gridZ,    // Grid dimension
3      blockX, blockY, blockZ, // Block dimension
4      0, null,                // Shared memory size and stream
5      parameters, null)     // Kernel- and extra parameters
6

```

Listing 4.7: CUDA kernel launching

The chosen dimensions used to test the system are:

```

1  int blockX = 1024;
2  int blockY = 1;
3  int blockZ = 1;
4
5  int gridX = ceil (L / 1024); // L = QUBO array length
6  int gridY = 1;
7  int gridZ = 1;

```

GridX is set to the length of the QUBO array L divided by $blockX$, that, in this case, is 1024. For a 100 variables matrix, the gridX dimension will be calculated as:

```

1   int N = 100;
2   int L = (100 * 101) / 2;
3
4   int blockX = 1024;
5   int gridX = ceil (L / blockX);

```

4.5 Testing and performances

The project has been implemented following the rules prescribed by the test-driven approach: this makes the diverse components of the system be implemented after their tests design and implementation. To verify the proper functioning of the project, Unit and Component tests have been set up throughout the framework [45]. In particular, Unit tests have been used to verify the correct behaviour of every single component while Component tests checked simultaneously more than one component and their interaction. Tests results and their coverage are shown in fig. 4.16, highlighting the fact that 73% of the classes have been tested. This percentage is related to the 100% of the model classes, while the remaining portion (27%) does not fit into the test. This portion, in fact, is composed only by utility classes that are not central in terms of the core system. Moreover it is clear that 98% of methods of the 73% classes and 96% of the code lines of the 98% methods have been also tested.

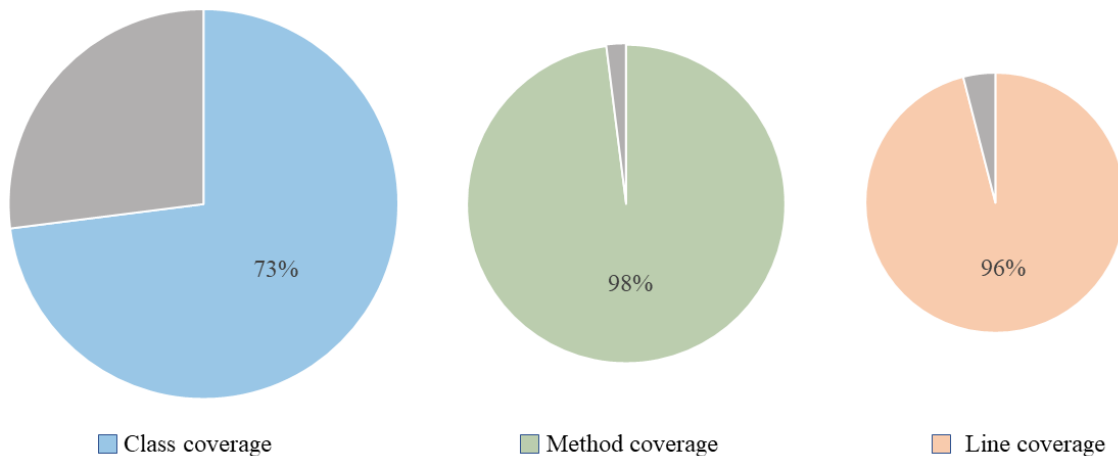


Figure 4.16: Test coverage

The disk space usage optimisation reached by using the row-major model to store a QUBO matrix of double numbers is reported in fig. 4.17. On the x axis there are the QUBO variables number and on the y axis the bytes used to store the matrices. The *complete* values represent the storing of the $N \times N$ matrix elements whilst the *row-major* values represent the storing of $(N \times (N + 1))/2$ matrix elements. In order to calculate the exact number of bytes used by the matrices the number of elements of each matrix has been multiplied by 8 because a single double number requires 8 bytes to be stored.

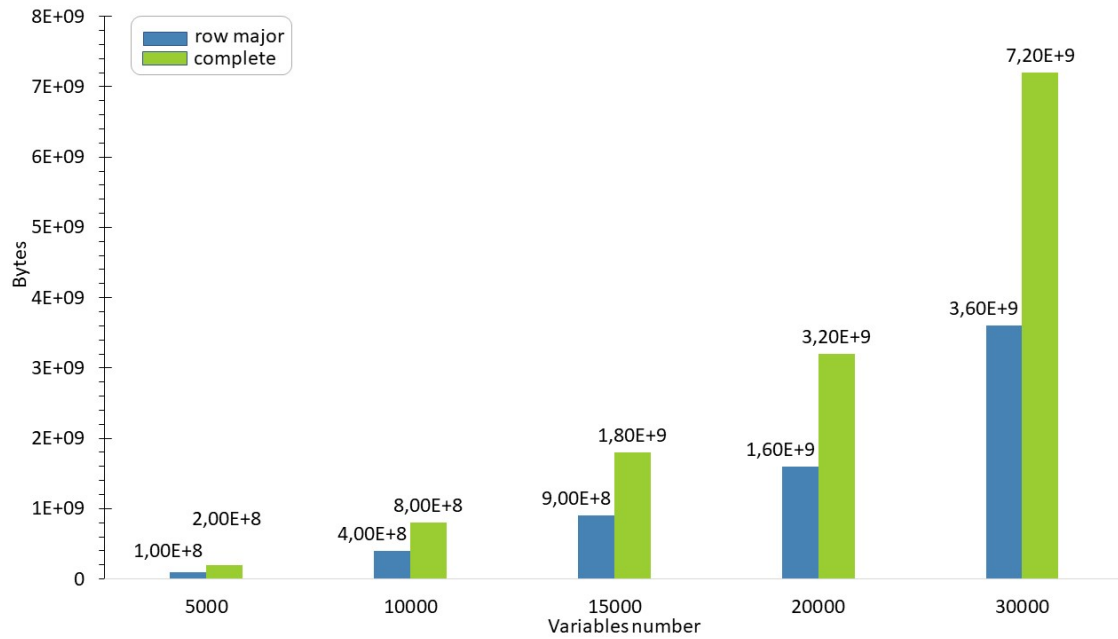


Figure 4.17: QUBO space usage

In order to assess the efficiency of the developed system, we used also a python script to simulate the functioning of previous system used by the company itself. This python code is reported in listing 4.8. Both the new project system, defined as GPU, and the python one, called CPU, were tested in the same application scenarios, creating QUBO matrices with an increasing number of variables (100, 1000, 5000, 10000, 15000, 20000 e 30000). All these matrices have been created with the same structure, primary costs set to 0 and two constraints:

- the assignment of any resource to only one job;
- the assignment of any job to only one resource.

```

1 res = resources_number
2 jobs = jobs_number
3 var = res * jobs
4 mat = np.zeros((var, var))
5
6 for i in range(0, res):
7
8     ith_node = np.zeros((jobs, jobs))
9
10    # C1: each resource can get only one job
11    constr_comp = 2 * np.ones((jobs, jobs))
12    np.fill_diagonal(constr_comp, -1)
13    ith_node += np.triu(constr_comp)
14
15
16    for j in range(i+1, res):
17
18        ij_node = np.identity(jobs)
19
20        # C2: each job can be assigned to only one resource
21        ith_node = np.concatenate((ith_node, ij_node), axis=1)
22
23    # QUBO matrix updating
24    mat[i*jobs:(i+1)*jobs, i*jobs:] = ith_node

```

Listing 4.8: Python script for creating QUBO models

These particular constraints involve the matrices shown in fig. 4.18. Models involved in real problems resolution are more complex than the model represented by the simple base matrices in the example mentioned above.

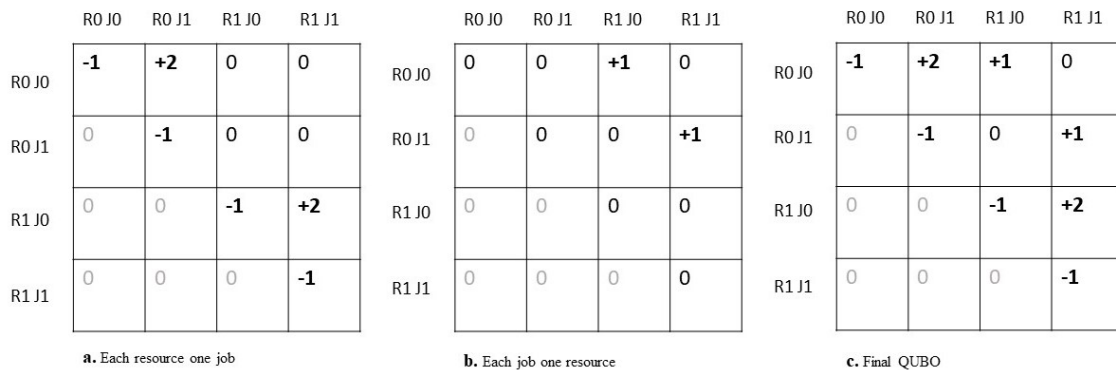


Figure 4.18: Testing QUBO

All the performed tests have been run on the same machine, an ASUS FX505G equipped with an i7-875H Intel processor and an Nvidia CTX GEFORCE GPU. In order to have reliable statistics, for any $N \times N$ matrix, both systems evaluated the constrained QUBO 50 times; the average times of all the results are provided in fig. 4.19.

The performances are reported in fig. 4.19 bear witness to the dependence between the elapsed time and the matrices dimensions. On the other hand, the GPU system

presents an increase in terms of computing time but, as stated in advance, an increase in the range of a few milliseconds borne by the GPU is not directly related to the matrix dimension but to the number of thread blocks to instantiate on the GPU. In fact, for a 1000×1000 matrix, the time the CPU takes to work on it is considerably higher than for a 100×100 matrix. For the GPU system the time remains almost the same, regardless of the matrix dimensions. The following diagram allows us to deduce that the new system performances are, at the best of times, up to four orders of magnitude better than that of the CPU system.

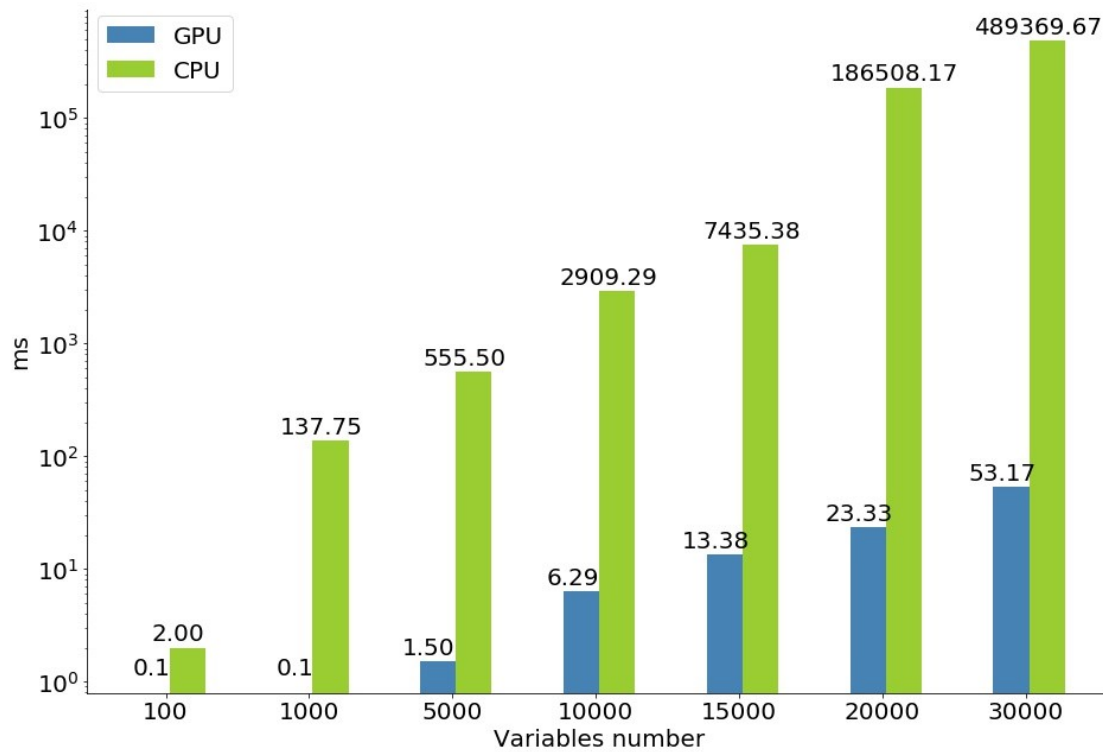


Figure 4.19: Test results

4.6 Future developments

The developed system, at present, is only a prototype to be used in simple examples. By extending some of its features, it will be able to make it usable within a more realistic environment. The main possible developments we identified are:

- **creation of a microservice:** the system will be wrapped on a microservice. This step is fundamental in as much as microservices permit to create independent application, to which you can accede through precise APIs. This would facilitate a machine-to-machine (M2M) interaction with other applications and the use of third-party functionalities from existing systems. The microservice will probably exhibit two endpoints:
 - a POST with which it receives the problem information;
 - a GET with which it supplies the proper QUBO matrix.
- **Interprocess communication (IPC):** using IPC techniques provided by CUDA would be an affordable and smart way to bypass the matrix on-disk saving. By doing so, the system would be considerably enhanced in terms of time because it would be no longer necessary transfer the QUBO matrix from GPU to CPU. Creating the matrix on the GPU and sharing the memory address of the matrix would only be necessary to use it. Of course, the processes the system has to interact with should be run on the same machine, so the GPU memory could be shared. However this seeming limitation should not result as a vital problem: in fact, running any process that requests to accede to the matrix on the same server should ease this drawback;
- **OpenCL:** a useful expansion for this system is represented by the possibility of choosing OpenCL or CUDA as programming languages for the creation of the QUBO matrix. This would permit to keep the best computation speed when working on Nvidia GPUs and also enables the computation on other kinds of GPU (i.e. AMD).

Chapter 5

Conclusions

The goal of the QoG project is to design and implement a system to create in the fastest way a QUBO model related to a combinatorial optimisation problem. Then this model can be solved through a Quantum Annealer that quickly finds the optimal solution for the initial problem, thanks to the properties of the Quantum Annealer itself.

Merging the programming language CUDA with a Scala system is possible to combine the optimisation given by the use of the GPU and the paradigm of object oriented programming. In this way, the resulting system is made capable of exploiting the GPU computing power and being readable and maintainable at the same time. The design part has been presented in detail in the section 4.2 section.

The system has been initially tested on a simple example of QUBO model, which consists of two recurrent constraints in several optimisation cases:

- the association of a resource to only one job;
- the association of a job to only one resource.

A matrix representation of this model can be found in fig. 4.18. Despite the apparent simplicity of this model, the tests we conducted gave us important results. By comparing the performances of the QoG with the Python system, as reported in fig. 4.19, we notice that the first one is much faster than the second one: for 30000×30000 matrices results differ by 4 orders of magnitude. Considering the standpoint of performances, the goal has been fully achieved.

The second goal was the optimisation in terms of space required to store and save the model to the device. As stated in section 4.2.3 section, by virtue of saving the QUBO matrix using a row-major model, it was possible to memorise only the portion with entries of the matrix, saving the memory needed in case the whole matrix was to be stored. The graph 4.17 shows what has just been affirmed and compares the memory usage of a whole matrix with a row-major modified matrix. In any case, the saving memory is approximately the 50% of the space required to the whole matrix. Therefore, the second goal has been achieved too. The main QoG future development is the creation of a microservice that wraps the system in order to simplify the interaction and the integration with other systems. A second important development is the integration of an IPC into QoG so it can be used to avoid the saving of the QUBO matrix on the filesystem. Once the QoG system was tested on

different base cases, it has been used in several other projects developed by the Data Reply Quantum team. In any case, the integration of a new kernel was very easy and quick, thanks to the usage of pattern factory. The speedup of the creation of QUBO model has been a crucial point for the resolution of real optimisation problems.

Glossary

AI	Artificial Intelligence.
API	Application Programming Interface.
AQC	Adiabatic Quantum Computing.
BQM	Binary Quantum Model.
CPU	Central Process Unit.
CUDA	Compute Unified Device Architecture.
GDDR	Graphics Double Data Rate.
GPGPU	General-purpose computing on graphics processing units.
GPU	Graphics Processing Unit.
ICT	Information and Communication technology.
IPC	Interprocess communication.
M2M	Machine to Machine.
MIMD	Multiple Instruction Multiple Data.
MPI	Message Passing Interface.
QA	Quantum Annealing.
QAE	Quantum Application Environment.
QC	Quantum Computing.
QMI	Quantum Machine Instruction.
QoG	QUBO on GPU.

QPU	Quantum Process Unit.
qubit	Quantum bit. It is the fundamental data unit for a quantum computer.
QUBO	Quadratic Unconstrained Binary Optimisation.
SDK	Software Development Kit.
SDRAM	Synchronous DRAM.
SIMD	Single Instruction Multiple Data.
SIMT	Single Instructions Multiple Thread.
SM	Streaming Multiprocessors.
SP	Stream Processors.
SPMD	Single Programming Multiple Data.

Bibliography

- [1] URL: <https://www.dwavesys.com/quantum-computing>.
- [2] Scott Aaronson. *Quantum Computing since Democritus*. Cambridge, 2013. ISBN: 9780521199568.
- [3] *alphaQUBO platform*. URL: <http://meta-analytics.net/Home/AlphaQUBO>.
- [4] Luca Asproni. “Quantum machine learning and optimisation: approaching real-world problems with a quantum coprocessor”. Politecnico di Torino, 2019.
- [5] Chris Bernhardt. *Quantum Computing for everyone*. MIT Press, 2019. ISBN: 9780262039253.
- [6] Fabio Chiarello. *L’officina del meccanico quantistico*. Maggioli editore, 2014. ISBN: 9788891602640.
- [7] Mark W. Coffey. “Adiabatic quantum computing solution of the knapsack problem”. In: (2017), p. 22.
- [8] Dr. Gabriele Compostella. “Quantum Computing at Volkswagen: Traffic Flow Optimization using the D-Wave Quantum Annealer”. In: D-Wave, Volkswagen. 2017, p. 23.
- [9] Shane Cook. *CUDA programming: A developer’s guide to parallel computing with GPUs*. 1st ed. Applications of GPU computing. Morgan Kaufmann, 2012. ISBN: 0124159338.
- [10] *CUDA Documentation*. URL: <http://www.jcuda.org>.
- [11] *CUDA thread indexing samples*. URL: <https://cs.calvin.edu/courses/cs/374/CUDA/CUDA-Thread-Indexing-Cheatsheet.pdf>.
- [12] D-Wave. *Getting Started with the D-Wave System*. English. D-Wave. 2018. 39 pp.
- [13] D-Wave. *PRACTICAL QUANTUM COMPUTING D-Wave Technology Overview*. D-Wave. 2020. URL: https://www.D-Wavesys.com/sites/default/files/D-Wave_Tech200verview2_F.pdf.
- [14] *D-Wave Applications*. URL: <https://www.dwavesys.com/applications>.
- [15] *D-Wave Ising, QUBO, and BQMs*. URL: https://docs.ocean.dwavesys.com/projects/dimod/en/latest/reference/bqm/binary_quadratic_model.html.
- [16] *D-Wave Map Coloring Problem*. URL: https://docs.dwavesys.com/docs/latest/c_handbook_1.html.

- [17] *D-Wave Minor Embedding*. URL: <https://docs.ocean.dwavesys.com/en/stable/concepts/embedding.html>.
- [18] *D-Wave Ocean documentation*. URL: <https://ocean.D-Wavesys.com/>.
- [19] *D-Wave QUBO Isign models*. URL: https://docs.D-Wavesys.com/docs/latest/c_handbook_3.html.
- [20] *D-Wave software overview*. URL: <https://www.D-Wavesys.com/software>.
- [21] *D-Wave Topologies*. URL: <https://docs.ocean.dwavesys.com/en/stable/concepts/topology.html>.
- [22] *D-Wave Using QUBOs to Represent Constraints*. URL: https://docs.dwavesys.com/docs/latest/c_gs_6.html.
- [23] *D-Waves documentation 1*. URL: <https://docs.D-Wavesys.com/docs/latest>.
- [24] Wen-Mei W Hwu David B. Kirk. *Programming Massively Parallel Processors: A Hands-On Approach*. 1st ed. Morgan Kaufmann, 2016. ISBN: 0128119861.
- [25] Mete Yurtoglu Duane Storti. *CUDA for Engineers. An Introduction to High-Performance Parallel Computing*. Addison Wesley, 2016. ISBN: 978-0-13-417741-0.
- [26] Ralph Johnson Erich Gamma Richard Helm and John Vlissides. *Design Patterns Elements of Reusable Object-Oriented software*. 2nd ed. Addison-Wesley, 1994. ISBN: 978-0-201-63361-0.
- [27] Yu Du Fred Glover Gary Kochenberger. “Quantum Bridge Analytics I: A Tutorial on Formulating and Using QUBO Models”. In: 6 (2019), p. 46. DOI: <https://arxiv.org/abs/1811.11538>.
- [28] John Gribbin. *Computing with quantum cats, from Alan Turing to teleportation*. Black Swan, 2013. ISBN: 9780552779319.
- [29] Lov K. Grover. “A fast quantum mechanical algorithm for database search”. In: (1996), p. 8. DOI: <https://arxiv.org/pdf/quant-ph/9605043.pdf>.
- [30] Cay S. Horstmann. *Scala for the Impatient*. 2nd ed. Addison-Wesley, 2016. ISBN: 0134540565.
- [31] Richard Hua. “Adiabatic Quantum Computing with QUBO Formulations”. University of Auckland, 2016.
- [32] Edward Kandrot Jason Sanders. *CUDA by Example: An Introduction to General-Purpose GPU Programming*. 1st ed. 2010. ISBN: 0131387685.
- [33] *JCUDA Samples repository*. URL: <https://github.com/jcuda/jcuda-samples>.
- [34] Jie Zhu, Hai Jiang, Juanjuan Li, Erikson Hardesty, Kuan-Ching Li, Zhongwen Li. “Embedding GPU Computations in Hadoop”. In: *International Journal of Networked and Distributed Computing* 4 (2014), p. 10. DOI: <https://www.atlantis-press.com/journals/ijndc/14323>.

- [35] Jack Raymond Kelly Boothby Paul Bunyk and Aidan Roy. *Next-Generation Topology of D-Wave Quantum Processors TECHNICAL REPORT*. 2019. URL: https://www.dwavesys.com/sites/default/files/14-1026A-C_Next-Generation-Topology-of-DW-Quantum-Processors.pdf.
- [36] Andrew Lucas. “Ising formulations of many NP problems. *Frontiers in Physics*”. In: (2014).
- [37] Bill Venners Martin Odersky Lex Spoon. *Programming in Scala*. 1st. Artima Inc, 2008. ISBN: 9780981531601.
- [38] N. David Mermin. *Quantum Computer Science An introduction*. Cambridge, 2007. ISBN: 9780521876582.
- [39] Isaac L. Chuang Michael A. Nielsen. *Quantum computing and quantum information*. 10th ed. Cambridge, 2016. ISBN: 9781107002173.
- [40] Florian Neukart et al. “Traffic flow optimization using a quantum annealer”. In: 2 (2017), p. 17. DOI: <https://arxiv.org/abs/1708.01625>.
- [41] NVIDIA. *NVIDIA CUDA C Programming Guide*. English. Version 9.1. NVIDIA. 2018. 301 pp. URL: https://docs.nvidia.com/cuda/archive/9.1/pdf/CUDA_C_Programming_Guide.pdf.
- [42] David Petrosyan Peter Lambropoulos. *Fundamentals of quantum optics and quantum information*. 1st ed. Springer, 2007. ISBN: 9783540345718.
- [43] *Quantum Computing Appunti delle lezioni*. URL: <http://profs.sci.univr.it/~dipierro/InfQuant/articles/Lezioni-IQ.pdf>.
- [44] *Quantum Annealing*. URL: https://medium.com/@quantum_wa/quantum-annealing-cdb129e96601.
- [45] *Scala test documentation*. URL: <https://www.scalatest.org/>.
- [46] Peter W. Shor. “Polynomial-Time Algorithms for Prime Factorization and Discrete Logarithms on a Quantum Computer”. In: (1996), p. 28. DOI: <https://arxiv.org/pdf/quant-ph/9508027.pdf>.
- [47] *Solving Max Cut problem using QUBO*. URL: https://tc3-japan.github.io/DA_tutorial/tutorial-3-max-cut.html.
- [48] Daniel A. Lidar Tameem Albash. “Adiabatic Quantum Computing”. In: (2018), p. 71. DOI: <https://arxiv.org/pdf/1611.04471.pdf>.
- [49] Google AI Teams. “Quantum supremacy using a programmable superconducting processor”. In: *Nature* (2019), p. 7. DOI: <https://www.nature.com/articles/s41586-019-1666-5.pdf>.
- [50] Cyrill Zeller. *CUDA C/C++ Basics*. English. NVIDIA. 2011. 68 pp.
- [51] William G. Macready Zhengbing Bian Fabian Chudak and Geordie Rose. “The Ising model: teaching an old problem new tricks”. In: (2010), p. 60. DOI: https://www.D-Wavesys.com/sites/default/files/weightedmaxsat_v2.pdf.