UNIVERSIDADE FEDERAL DE GOIÁS
INSTITUTO DE INFORMÁTICA

WALID ABDALA RFAEI JRADI

Application of GPU Computing to Some Urban Traffic Problems

Goiânia
2016

sistema de bibliotecas ufg

UFG

TERMO DE CIÊNCIA E DE AUTORIZAÇÃO PARA DISPONIBILIZAR AS TESES E DISSERTAÇÕES ELETRÔNICAS NA BIBLIOTECA DIGITAL DA UFG

Na qualidade de titular dos direitos de autor, autorizo a Universidade Federal de Goiás (UFG) a disponibilizar, gratuitamente, por meio da Biblioteca Digital de Teses e Dissertações (BDTD/UFG), regulamentada pela Resolução CEPEC nº 832/2007, sem ressarcimento dos direitos autorais, de acordo com a Lei nº 9610/98, o documento conforme permissões assinaladas abaixo, para fins de leitura, impressão e/ou *download*, a título de divulgação da produção científica brasileira, a partir desta data.

1 **1. Identificação do material bibliográfico: [ ] Dissertação [X] Tese**

1 **2. Identificação da Tese ou Dissertação**
2

Nome completo do autor: Walid Abdala Rfaei Jradi

Título do trabalho: Application of GPU Computing to Some Urban Traffic Problems

3. Informações de acesso ao documento:

Concorda com a liberação total do documento [X] SIM [ ] NÃO[1]

Havendo concordância com a disponibilização eletrônica, torna-se imprescindível o envio do(s) arquivo(s) em formato digital PDF da tese ou dissertação.

Walid Abdala Rfaei Jradi

Assinatura do (a) autor (a)

Data: 30 / 12 / 2016

[1] Neste caso o documento será embargado por até um ano a partir da data de defesa. A extensão deste prazo suscita justificativa junto à coordenação do curso. Os dados do documento não serão disponibilizados durante o período de embargo.

WALID ABDALA RFAEI JRADI

Application of GPU Computing to Some Urban Traffic Problems

Tese apresentada ao Programa de Pós–Graduação do Instituto de Informática da Universidade Federal de Goiás, como requisito parcial para obtenção do título de Doutor em Computação.

Área de concentração: Ciência da Computação.

Orientador: Prof. Hugo Alexandre Dantas do Nascimento

Co-Orientador: Prof. Wellington Santos Martins

Goiânia
2016

Ficha de identificação da obra elaborada pelo autor, através do
Programa de Geração Automática do Sistema de Bibliotecas da UFG.

Jradi, Walid
Application of GPU Computing to Some Urban Traffic Problems [manuscrito] / Walid Jradi. - 2016.
CXCII, 192 f.: il.

Orientador: Prof. Dr. Hugo Nascimento; co-orientador Dr. Wellington Martins.
Tese (Doutorado) - Universidade Federal de Goiás, Instituto de Informática (INF), Programa de Pós-Graduação em Ciência da Computação, Goiânia, 2016.
Bibliografia. Apêndice.
Inclui gráfico, tabelas, algoritmos, lista de figuras, lista de tabelas.

1. Urban Traffic. 2. Macroscopic Traffic Allocation. 3. Parallel Computing. 4. GPU. I. Nascimento, Hugo, orient. II. Título.

CDU 004

UNIVERSIDADE FEDERAL DE MATO GROSSO DO SUL
UNIVERSIDADE FEDERAL DE GOIÁS
INSTITUTO DE INFORMÁTICA
DOUTORADO EM CIÊNCIA DA COMPUTAÇÃO

UFMS

Ata de Defesa de Tese de Doutorado

Aos trinta dias do mês de novembro de dois mil e dezesseis, no horário das nove horas, foi realizada, nas dependências do Instituto de Informática da UFG, a defesa pública da Tese de Doutorado do aluno Walid Abdala Rfaei Jradi, matrícula no. 2010 1191, intitulada **"Application of GPU Computing to Urban Traffic Problems"**.

A Banca Examinadora, constituída pelos professores:

Prof. Dr. Hugo Alexandre Dantas do Nascimento - (INF/UFG) – orientador,

Prof. Dr. Wellington Santos Martins – INF/UFG - coorientador

Prof. Dr. Eduardo Camponogara – DAS/UFSC

Prof. Dr. Esteban Walter Gonzalez Clua – IC/UFF

Prof. Dr. Henrique Mongelli – FACOM/UFMS

Prof. Dr. Fábio Moreira Costa – INF/UFG

emitiu o resultado:

(X) Aprovado

( ) Aprovado com revisão

(A Banca Examinadora deve definir as exigências a serem cumpridas pelo aluno na revisão, ficando o orientador responsável pela verificação do cumprimento das mesmas.)

( ) Reprovado com o seguinte parecer:

Prof. Dr. Hugo Alexandre Dantas do Nascimento

Prof. Dr. Wellington Santos Martins

Prof. Dr. Eduardo Camponogara

Prof. Dr. Esteban Walter Gonzalez Clua

Prof. Dr. Henrique Mongelli

Prof. Dr. Fábio Moreira Costa

Walid Abdala Rfaei Jradi

Walid Abdala Rfaei Jradi received his bachelor degree in Data Processing from the Instituto Unificado de Ensino Superior (IUESO) in 2005 and his master degree in Computer Science in 2008. During his graduation, he developed a study on modeling and simulation of sewage and rainwater collector networks. He also developed studies on the traveling salesman problem to optimize a system for the distribution of goods for a transportation company. In his master degree, he proposed and developed a Web-Based traffic simulator suited to the way traffic behaves in Brazilian urban road networks. During his Ph.D., in addition to the study here presented, he also worked on problems related to two-dimensional guillotine cutting, as well as acting in the development of the PET-Gyn software version 2.0.

To the authentic seekers of the truth that, even on their tireless search, still find time to laugh and marvel of this crazy journey we call life.

Acknowledgments

To God, for the gift of life, and for allowing me to continue this trip.

To my family. The unconditional support of my parents, Abdala and Inaam, my brothers Tarek and Hanan and my brother in law Eduardo was the cornerstone of this achievement, and is the light that guides me upon my darkest days.

To my beloved grandmother, who unfortunately did not live long enough to witness this achievement, but never doubted that I would be able to accomplish this work. Rest in peace, because you were the best grandmother that anyone could wish for.

To Professor Hugo Nascimento, guide of this work. The fact that he believed that I would be able to complete this project, even when I doubted, reveals a nobility of spirit hard to find.

To my co-advisor, Professor Wellington Martins, for the patience and willingness to solve all my questions, even the seemingly unjustified ones.

To the friends who I had conquered and the new ones along the way, especially Elisângela Dias, Halley Gondim, Luciana Berretta, Márcia Cappelle, Márcio Duarte, Roussian Gaioso, Rafael Quirino, Jesmmer Alves and Wanderley Alencar, always solicitous to help me at any time. To all, my sincere gratitude.

To Professors Humberto Longo, Bryon Hall, Diane Castonguay and Les Foulds. Adapting the famous phrase, “If I rise so high it is by standing on the shoulders of giants”.

To Professor Cláudio Maia, who extended his hand in a crucial moment of the work. It’s amazing how the opinion of someone a lot cleverer than me can be helpful.

To CAPES (Coordenação de Aperfeiçoamento de Pessoal de Nível Superior) for providing a PhD scholaship that supported the present research. And to FAPEG (Fundação de Amparo à Pesquisa do Estado de Goiás), for the financial resources necessary for the development of PET-Gyn, version 2.0.

To Wilk Araújo. His friendship and care were of fundamental importance to overcome my physical limitations.

And to Fabrizzio Soares. Without his encouragement, it would not be possible to engage in this insane adventure that today is complete.

To all, my most sincere “Thank you”.

Never lose hope, even facing the worst sorrows of your life. From the darkest clouds clean and refreshing water drops.

Chinese Proverb,
Unknown author.

Resumo

Jradi, Walid Abdala Rfaei. **Application of GPU Computing to Some Urban Traffic Problems**. Goiânia, 2016. 192p. Tese de Doutorado Relatório de Graduação. Instituto de Informática, Universidade Federal de Goiás.

O presente trabalho estuda e propõe algoritmos e implementações paralelas baseadas em GPU para o problema de alocação macroscópica de tráfego urbano em redes de grande porte, promovendo uma investigação aprofundada de cada sub-problema que deve ser resolvido de forma eficiente durante o processo de atribuição de tráfego. Entre as principais contribuições deste trabalho, estão: 1) o primeiro algoritmo baseado em GPU para a enumeração de ciclos sem corda; 2) um novo algoritmo de caminho mínimo paralelo que tira vantagem de algumas propriedades comuns das redes de tráfego urbano; Um refinamento na implementação de redução paralela proposta por um dos líderes no mercado de GPU, o que resultou em uma aceleração de 2,8x em relação à sua versão original; 3) e, finalmente, um algoritmo paralelo para o problema de alocação macroscópica de tráfego, 39x mais rápido do que a abordagem equivalente sequencial quando aplicado a redes de larga escala.

O objetivo principal desta tese é de contribuir para a expansão do software PET-Gyn, propondo estruturas de dados de GPU eficientes e algoritmos paralelos para uma resolução mais rápida de dois problemas bem conhecidos na literatura: O *Problema de Alocação de Tráfego* e a *Enumeração de Ciclos sem Corda*. Quando aplicados a conjuntos de entrada difíceis, os experimentos realizados mostraram uma clara vantagem dos algoritmos paralelos sobre suas versões sequenciais.

Palavras–chave

Tráfego Urbano, Alocação Macroscópica de Tráfego, Computação Paralela, GPU.

Abstract

Jradi, Walid Abdala Rfaei. **Application of GPU Computing to Some Urban Traffic Problems**. Goiânia, 2016. 192p. PhD. Thesis. Instituto de Informática, Universidade Federal de Goiás.

The present work studies and proposes GPU-based parallel algorithms and implementations for the problem of macroscopic assignment of urban traffic on large-scale networks, promoting an in-depth investigation on each sub-problem that must be efficiently solved during the traffic assignment process. Among the main contributions of this work, there are: 1) the first GPU-based algorithm for the enumeration of chordless cycles; 2) a new parallel GPU-based shortest path algorithm that takes advantage of some common properties of urban traffic networks; a refinement in the parallel reduction implementation proposed by one of the leaders in the GPU market, which resulted in a 2.8x speedup relative to its original version; and finally, 3) a parallel algorithm for the macroscopic traffic assignment problem, 39x faster than the equivalent sequential approach when applied to large scale networks.

The main goal of this thesis is to contribute to the extension of the PET-Gyn software, proposing efficient GPU data structures and parallel algorithms for a faster resolution of two well known problems in the literature: The *Traffic Assignment Problem (TAP)* and the *Enumeration of Chordless Cycles*. When applied to difficult input sets, the performed experiments showed a clear advantage of the parallel algorithms over their sequential versions.

Keywords

Urban Traffic, Macroscopic Traffic Assignment, Parallel Computing, GPU.

Contents

List of Figures

List of Tables

List of Algorithms

Listings

CHAPTER 1

Introduction

1.1 Motivations

Road traffic in large Brazilian cities is, in most urban centers, in a chaotic state. The phenomenon occurs mainly due to poor design of the road network, when compared to the number of vehicles that travel on public roads. It is also worsened by the constantly growing need for displacement of people and goods, which mostly have origin in economic activities, social interaction and recreation.

This constant increase, coupled with the chronic lack of adequate government policies for the sector inexorably lead to the deterioration of the space in the road network available for the flow of vehicles, multiplying the bottlenecks and the time required for the completion of travel.

According to a report from IPEA – Instituto de Pesquisa Econômica Aplicada, or Institute for Applied Economic Research in english – the delay of drivers stuck in traffic jams results in estimated economic losses of R$ 26.8 billion/year just in the city of São Paulo [84]: "*After all, there are estimates pointing that people waste two to three hours a day in traffic. This means that in the course of a month they spent at least two days in the bus or car*" (translated from portuguese).

In their daily activities traffic engineers use, among other tools, mathematical models and computer simulations in order to predict driver behavior and estimate the impact of possible changes in the road network structure. Such simulation systems are valuable tools for the design and evaluation of transportation networks.

However, in real urban scenarios, it is often hard to ensure that changes[1] in the road network structure will effectively contribute to the improvement of traffic, under the considered aspects. The analysis and evaluation of projects that change the road network infrastructure should be made carefully and based on supporting data.

[1] Such as: reversal in the vehicles direction on a street, permission and/or prohibition of conversions in certain directions, opening/closing times of traffic lights, parking deny/permission, creation/removal of roundabouts, open/close/enlarge roads, creation of bridges, tunnels, etc.

A field of study called *Network Design Problem (NDP)* arose from this need. Its objective is to determine (among previously identified amendments, usually proposed by traffic engineers) which changes in the network infrastructure can provide the best result, when a number of aspects for analysis is considered and therefore have to be implemented.

In order to predict how urban traffic will behave according to the proposed network changes without having to actually implement them, usually a computer simulation called *Traffic Assignment* (a resolution of an instance of a *Traffic Assignment Problem (TAP)*) may be performed using a model of the modified network. With this supporting data, it is possible to assess whether the planned interventions are beneficial and should be implemented, or should instead be discarded. The way these two activities are connected is illustrated in Figure 1.1.

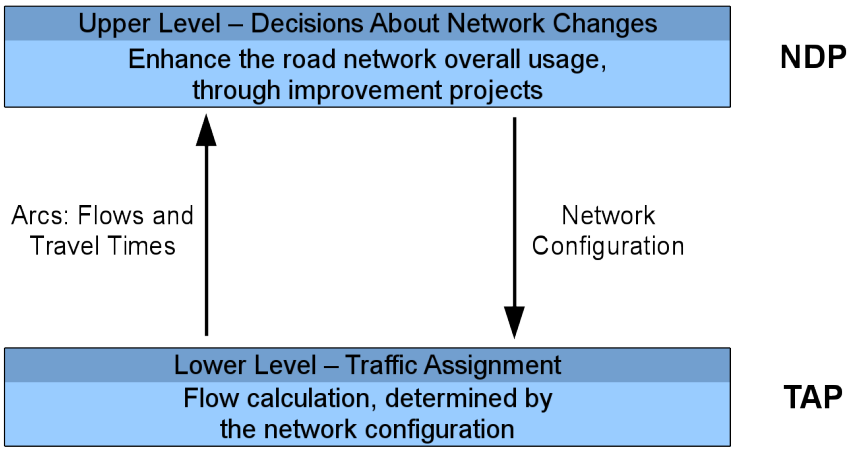


Figure 1.1: *Hierarchy between NDP and TAP.*

The computational implementation of a traffic simulator is a process that can be accomplished under several distinct mathematical approaches and, depending on the choice, it becomes less or more financiall and computationally costly. The level of detail of the simulation is the determining factor of these costs. There are essentially three major levels of detail: microscopic, mesoscopic and macroscopic which are described in detail in Chapters 2 and 8.

Encouraged by the growing interest in providing tools that aid traffic engineers in this decision-making processes, since 2006 the Federal University of Goiás (UFG) through its Informatics Institute and its Mathematics and Statistics Institute has developed a web-based urban traffic simulator called PET-Gyn [142][2]. It employs a macroscopic mathematical approach to estimate vehicle flows that is suitable to the Brazilian reality, when compared with aproaches used abroad. Such a system was submitted to experts

[2]The adopted nomenclature – PET-Gyn – is an acronym for Traffic Equilibrium Problem (or Problema de Equilíbrio de Tráfego, in Portuguese) applied to the city of Goiânia, Goiás, Brazil, whose international aeronautical acronym is GYN

who praised the initiative of developing technologies for this area, quite lacking in our country. They also acclaimed the importance of releasing the system on-line, allowing collaborative modeling, simulation and analysis of networks by traffic planners.

In subsequent years, PET-Gyn was used as a framework for other analysis tools, which focused on aspects not covered by the original research [36, 67, 78, 85, 99, 111, 112, 113]. This proved the robustness of the developed architecture and motivated the continuation of its development, now under a broader perspective and embracing new problems related to the area.

Therefore, an entire redesign of the system started in 2013, aiming at updating the PET-Gyn graphical user interface and also at making the software more flexible for incluiding new pieces of research, with its first 2.0 version been completed in March 2016.

One of PET-Gyn's key aspects is to provide traffic engineers with a supporting tool to help the analysis and evaluation of projects that change the road network infrastructure. However, given the restrictions in the computational power of the equipments usually available to traffic agencies, it is currently not feasible to carry out such restructuring analysis in large road networks (i.e., comprising avenues, vehicles, traffic signal units, etc.) of significant region of a metropolitan area. These involve several alternating phases of modification of the road network and subsequent resolution of a TAP until a valid and economically feasible proposal for improving the traffic is found.

One alternative to overcome the aforementioned issue is the use of parallel computing, since this technique increases the processing power of commercially available computers, therefore allowing the expansion of both the size of the problem in focus and the level of detail to be analyzed.

Nowadays, on individual machines (not interconnected by a local area network) such computational power can be achieved in two ways: by the use of computers equipped with multi-core microprocessors and the use of new, programmable and powerful video cards, known as *Graphic Processing Units – GPUs*.

Most approaches that deal with parallel traffic allocation focuses on micro and mesoscopic models, naturally quite expensive in computational terms. Unfortunatelly very little research has been conducted on parallelism in macroscopic models when applied to large road networks. There is also no GPU-based approaches for urban traffic macroscopic modeling.

1.2 Aims

In this thesis we study and propose parallel approaches based on GPUs for the analysis of urban traffic conditions. We focus specifically on two topics:

- The fast enumeration of chordless cycles in graphs, since the study of these structures can help traffic planners to identify regions of the urban traffic network that are poorly connected and, hence, can act as barriers to movement;
- The *Traffic Assignment Problem (TAP)*, that briefly refers to the assignment of vehicles on the road network according to its structural settings, as well as information about traffic volumes. A traffic assignment method allows the understanding of how travel demands become traffic flows in a network. This is very important for perceiving the causes of traffic congestions and of other traffic costs in some areas and for evaluating changes in the network that can alleviate these problems.

For the first problem, a GPU parallel method is implemented based on a new sequential approach described by Dias et al. [74]. Here, both execution time and memory space metrics are considered since, in this case, the memory consumption is critical due to the extremely fast way the number of chordless cycles grows with the increase of the size of the input graph.

For the second problem, a number of steps that composes a classical sequential traffic assignment method are studied and equivalent parallel algorithms using GPUs are designed in order to identify the best structures – of data and flow control – for improving their performance. We start with two sub-problems common in those steps, which are the execution of a reduction operation and computing shortest paths.

Despite considering memory space usage in the chordless cycle problem, the main performance metric in the present thesis is execution time. We assume that the amount of memory available in current GPUs[3] does not represent a limitation to the solution of the problem under study, if the correct data structures are chosen.

1.3 Research Methodology

In order to achieve the goals proposed by the study, the following steps have been specified:

1. Performing a literature review on parallel computing and GPU technology, among other relevant studies related to these topics;
2. Carrying out a literature review on traffic engineering and the models of traffic assignment (TAP), as well as parallel and distributed approaches used in this field of study;

[3]Commercial GPUs offer specialized memories that, in many cases, reach the level of several gigabytes. Models with 1GB, 2GB and 4GB are usual, reaching 32GB in models considered *state of the art*. In addition, there is the possibility of using multiple cards together, bringing the total memory available to the order of hundreds of gigabytes.

3. Conceiving, designing and implementing GPU-based parallel algorithms for each step of the TAP based on a well established sequential method;
4. Evaluating the implemented parallel algorithms for TAP;
5. Conceiving, designing and implementing a GPU-based parallel algorithm for the fast enumeration of chordless cycles in graphs;
6. Testing, reviewing and evaluating the implemented enumeration algorithm.

1.4 Contributions

The main contributions of the thesis are:

- The first GPU-based algorithm for the enumeration of chordless cycles;
- A refinement of the parallel reduction implementation proposed by one of the leaders in the GPU market, which resulted in a 2.8x speedup relative to its original version;
- A new parallel GPU-based shortest path algorithm that takes advantage of some common properties of urban traffic networks;
- A parallel algorithm for the macroscopic traffic assignment problem and an implementation that is 39x faster than the sequential equivalent approach when applied to large scale networks.

1.5 Organization of the Thesis

The remainder of this work is organized as follows: Chapter 2 provides a literature review of the basic concepts of urban traffic modeling. Chapter 3 summarizes basic concepts about parallel computing and the current state of tools and techniques for GPU programming. Chapter 4 describes some parallel systems for traffic simulation. Chapter 5 presents a GPU parallel method to enumerate all chordless cycles in a given graph. Chapter 6 shows a platform-independent and fast strategy to perform parallel reductions on programmable video devices. Chapter 7 depicts an efficient way to solve the single source shortest path problem on GPUs for urban traffic networks. Chapter 8 details the main concepts of macroscopic urban traffic assignment, the associated mathematical modeling and the implemented GPU parallel system. Finally, Chapter 9 draws the conclusions, the main contributions of the present thesis and proposals for future work.

CHAPTER 2

Urban Traffic Simulation Models

According to Cascetta [38], a *Transportation System* can be defined as a combination of elements and their interactions, which produce both travel demands and the provision of transport services to meet those demands. Such a definition is quite general and can be applied in different contexts. In general, *Transportation Engineering* deals with the design and evaluation of projects of transportation systems.

Traffic Engineering, in turn, is a specialization of Transportation Engineering, whose objectives are to plan – to define goals, steps, deadlines and means – the geometric design of roads and how they will relate to other means of transportation and the traffic operations in the road network as well. Traffic Engineering should always have the basic premises to guarantee a safe, efficient and convenient movement of people and goods at acceptable costs [199].

The aspect that distinguishes Traffic Engineering and differentiates it from other areas of Engineering is that it deals with issues that do not rely solely on physical agents, but often must also consider the behavior of drivers and pedestrians and their interactions with the surrounding environment.

While it may seem that the work of the authorities responsable for traffic is simple – because at a first glance they just have to increase network capacity in order to improve it – this is not always true. A famous example is given by the *Braess Paradox*. According to it, it is possible (though unlikely) that *"to increase the network capacity, creating a direct connection between two initial paths (a third way), has the effect of increasing the length of time for **all** system users"* [95], i.e., travel time will be **higher** than before the construction of the new road.

This little obvious phenomenon could be observed in practice in the cities of New York (United States) and Stuttgart (Germany). In 1990, the 42nd avenue of the city of New York was closed for the *Earth Day* and, despite predictions that it would create chaos, the traffic at that area actually improved [153]. The opposite occurred in Stuttgart: a new avenue was built in downtown and the traffic worsened significantly. The expected benefits were only achieved after a crossing avenue had been closed to traffic [19].

Having made these considerations, Traffic Engineering is characterized as a

multidisciplinary field of knowledge. Ideally, a study team should be composed by civil, structural and traffic engineers, landscape architects, urban planners, sociologists, urban geographers, economists, mathematicians (applied Mathematics), lawyers and market analysts [56].

This chapter describes and details the main mathematical models for urban traffic simulation and the assignment of vehicle flows on the road network. It is organized as follows: Section 2.1 lists the main models of traffic assignment. Section 2.1.1 provides some details on how the microscopic models operate and its computational costs. Section 2.1.2 briefly describes the mesoscopic models. Section 2.1.3 presents the macroscopic models and why they were chosen for use in this work. Section 2.2 gives some details on the macroscopic models, its basic definitions and the two main macroscopic models available in literature. Finally, Section 2.3 presents some general remarks about the present chapter.

2.1 Background

According to Ortuzar and Willumsen [192], a model can be understood as a simplified representation of a part of the real world, which seeks to focus on several elements considered important to the analysis, under a certain point of view. Thus, the modeling of traffic tries to describe its behavior using computational and mathematical resources, in order to better understand their problems or predict future behavior.

Traffic modeling is based on a hypothesis that, although imperfect, a model is useful since it corresponds to the majority of urban flows, especially at peak times: It is assumed that every driver is familiar with the road network and wants to minimize its own travel time (cost). It is also assumed that the displacement demand is fixed.

A myriad of traffic models can be found in the literature that try to reproduce the reality according to different approaches. These models, however, have many points in common, allowing their classification under some aspects. One of these is the *road flow representation levels* (also called *aggregation levels*), according to which, models can be categorized into micro, meso or macroscopic [62, 162], as described next.

2.1.1 Microscopic Models

Microscopic models simulate in detail the individual behavior of each vehicle such as acceleration, braking, lane change, etc., as well the consequences of each individual action in relation to the other vehicles. They operate stochastically (randomly generating various simulation parameters) and require a large computational effort and a large amount of data [106, 162]. Among the proposals that can be classified as microscopic a

number of methods can be cited, which are based on: Car-Following, Cellular Automata, Monte Carlo, Discrete Events and Continuous-Time Simulation.

Despite the high development and maintaining costs and the running difficulties of microscopic models [162], there is a considerable amount of academic and commercial applications that implement them, including systems such as AIMSUN2, DRACULA, CORSIM, SimTraffic, VISSIM, among others. A detailed description of these systems, a brief history as well as a comparative table of their main features can be found in [163, 206, 210].

More recently, efforts have been made toward an even greater separation of the elements involved in the traffic, considering individual behavior of drivers (with detailed modeling of perception, decisions and mistakes committed) and vehicles. This approach is called *nanoscopic* [6, 72, 154, 196].

Thanks to the microscopic model's inherent philosophy – which aims to estimate the real-time behavior of the vehicles traveling on the road network – and to the extent that the behavioral models are improved – with more and more aspects of the actual traffic being considered in the simulation environment – the computational complexity also rises significantly. Despite the steady increase in the processing power of modern personal computers, such a kind of simulation remains computationally expensive, limiting the maximum size of the network that can be simulated on a PC – even using parallel computing on a multi-processor machine – in reasonable time [246].

Trying to circumvent such limitations, software designers and engineers usually make use of distributed computing – be it on a local network or in a *Wide Area Network* (WAN) – in order to achieve efficient simulation of large areas and high volumes of vehicles [166, 246].

However, other implementation and execution problems arise when distributing the work between different nodes (or participants) of a network [165] because the algorithms and communication structures must be adapted for distributed environments. The first and most obvious difficulty is the extra layer of code that must be created for communication and synchronization between the network nodes. Taking into account that the communication environment is considerably slower than internal computer data traffic [77, 203], its use should be minimized.

Another problem appears when splitting the computation through nodes of a network. This task implies that the traffic system should be partitioned into roughly equivalent complexity subsystems to ensure a homogeneous workload among the participants. If this is not possible, the simulation performance can be severely degraded [223].

The partition process should also ensure that the division into subsystems occurs only in regions where there is no strong interaction between the avenue segments (i.e., it does not occur in the direct vicinity of path crossings), a task which is not always

trivial [83].

A less obvious – but not less important – impediment to the adoption of this solution is that many traffic agencies are not able to afford the high implementation and maintenance costs of a network exclusively dedicated to simulation and study of urban traffic.

2.1.2 Mesoscopic Models

Mesoscopic models are deterministic models that describe the system entities with a large number of details, but treat their activities and interactions (for example, a lane change) with much less precision when compared to microscopic models [24, 34, 162]. These models do not consider individual vehicles as elements of traffic, but the platoons they create during their displacement, according to the momentary interruptions in flows due to traffic lights. The mesoscopic models are quite useful, e.g., in the definition of traffic lights synchronization policies.

There are several computer programs and studies that implement hybrid models, combining micro and mesoscopic simulations, thanks to the proximity between them, such as the studies of Burghout et al. [32, 33, 34, 35], Yang et al. [254], Vilaró et al. [242] e Mammar et al. [169], among others.

2.1.3 Macroscopic Models

Macroscopic models treat traffic as a single entity in the form of an equilibrium system, describing it through relationships between flow, density and speed [62]. These models generally represent the traffic streams as a continuous fluid flowing in a similar way to liquids, basing its theoretical formulation upon the laws of hydrodynamics [201, 235].

They are the less expensive in computational and implementation terms, but are also the most inaccurate. Because of this imprecision, their use is traditionally justified when [106, 162]:

- the calculation result is not sensitive to microscopic details;
- the scale of the simulation does not allow the execution of microscopic models, due to their high execution time; and
- resources and time available for the construction of the application are limited.

On the other hand, it has been observed that the uncertainties involved in the data survey of road flows make the concern for accuracy less relevant when searching for the optimal solution [5]. This has caused the macroscopic models to grow in popularity to

the point that they are now widely used in simulations of urban traffic. Thus, the present study chooses a macroscopic model as the traffic assignment model.

2.2 Details on Macroscopic Models

A *road network* is a simplified representation of an existing road mesh structure [200]. To measure the behavior of vehicles on a road network, it has to be submited to a traffic circulation, being such activity called *traffic assignment* or *traffic allocation*. In this process, some simplifying assumptions are made about road traffic:

- the vast majority of drivers travels from a source to a destination;
- the vast majority of drivers knows the local geography and what possible paths are the most economical at that time[1].

The macroscopic assignment models have as input data, besides the urban road network structure, rules for route selection, as well as a set of demands that specify the amount of traffic between origin-destination [87] pairs. As output data, these models generate the averages of vehicle flow and travel time on each road.

Next we present the basic formal definitions of macroscopic modeling and how the traffic assignment process is computed.

2.2.1 Basic Definitions

Be the *road network* $G = (V,E)$, where V is a set of nodes (intersections) and E is a set of directed arcs (streets or avenues). Each arc $a \in E$ has a length (in meters) c_a, a number of traffic lanes[2] f_a and a free speed[3] v_a.

A *demand* $d_{i,j} \in \mathcal{V}, d_{i,j} > 0$ corresponds to the number of vehicles that intend to move from i to j, respectively origin and destination nodes in the road network G. When the source and destination points are implied, the demand is represented only by d.

A vector ($\mathcal{D}$) of demands of vehicles between origins and destinations ($\mathcal{OD}$) in G is considered to be known. This vector usually contains demands for a certain period of the day.

[1] This assumption may not be entirely correct in some cases. However, it is not a complicating element of the computational model, since it is possible to associate a maximum free speed slightly smaller than the true one for a little-known urban way. This minimizes the use of these routes by the shortest path algorithms, effectively simulating the fact that they are not well known.

[2] The width of the street/avenue lane is measured without parked vehicles. Each 3.5 meters in width corresponds to a lane.

[3] Average speed maintained by the drivers in a situation of completely free road or with small flow – in km/h.

A *flow* x_a in arc a is the amount of vehicles that travel along it during a certain period of time. The *network flow* X_G is given by $X_G = \bigcup_{a \in E} x_a$. When computed, X_G is considered viable if it meets the total demand ($\mathcal{D}$).

X_d represents the *demand's flow* $d \in \mathcal{D}$, with $X_d \in \mathcal{R}^{|E|}$. In other words, X_d consists of all arc flows of the network necessary to meet the specific demand d, including the null flows.

The *arc cost*, represented by t_a, is the average travel time of the vehicles using this arc. Such a cost is given by latency functions based on the previous parameters (c_a, f_a and v_a) and on the amount of vehicle flow through a street/avenue. As usually a road network is composed of several kinds of streets and avenues, for each kind there is a specific arc cost function.

The traffic network time T_G is the vector of times t_a required to travel at all arcs $a \in E$, with $T_G \in \mathcal{R}^{|E|}$ and $T_G = \sum_{a \in E} t_a$.

The *road network state* is given by $S_G =< X_G, T_G >$ and the *traffic total cost* is given by $C_G = \sum_{a \in E} t_a * x_a$.

The *Traffic Assignment Problem (TAP)*, therefore, consists of, given G, functions for t_a and D, to find a feasible flow X^* such that any feasible flow X is worse than X^*.

2.2.2 User Equilibrium and System Optimization

The first suggestion of considering the urban traffic as a system in equilibrium was made in 1924 by Frank Knight [151]. Later, in 1952, Wardrop [247] established two principles using this concept in order to formalize the notion of urban traffic in an equilibrium condition: the *Wardrop's Principles*.

According to his definitions, under equilibrium the traffic generally tries to reach one of the following two status [127]:

- **User Equilibrium (UE):** Here, the traffic conforms to the *Wardrop's First Principle* [247]. This principle states that, under equilibrium conditions, traffic in congested networks self-organizes so that all used routes between a source and destination pair have minimal cost, while all unused routes have higher costs than those ones. Drivers individually choose their faster routes, and the system enters in equilibrium when no vehicle can improve its own travel time through an individual route change;
- **System Optimization (SO):** Wardrop also proposed an alternative form of traffic assignment in a transportation system, known as *Wardrop's Second Principle* [247]. In this state the transportion system is arranged (probably by an all-powerful central intelligence) in a way that the sum of the travel times of all drivers is the smallest possible.

The UE condition tries to reproduce the actual drivers' behavior while SO depicts an ideal behavior, in which the traffic system is globally optimized, sometimes to the detriment of individual drivers' interest. The ***T**raffic **E**quilibrium **P**roblems (TEP)* [57] that focus on a traffic assignment under the UE and SO conditions are called, respectively, TEP-UE and TEP-SO.

The *price of anarchy* – term formally introduced in [155] – is a relationship between the solutions for TEP-UE and TEP-SO. The comparison of these two solutions indicates how far from ideal the network under analysis is, indicating the intrinsic level of inefficiency of the current road network, also being an estimate of the impact of individual drivers decisions on the whole system [47].

2.2.3 Beckmann's Model

Beckmann et al. [11] were the first ones to present a mathematical model[4] to compute both TEP-UE and TEP-SO. Since then, their approach became the *de facto* macroscopic model [25, 47, 182]. In their model, the central point is the formulation of latency functions (in the present study, the average travel time t_a) for each street/avenue of the road network under analysis.

These functions (which must meet the requeriments of convexity, continuity, monotonic increase and strictly positive) have to be defined in such a way that, as more vehicles simultaneously use the arc, its latency also increases proportionally, making it progressively less attractive and forcing users to choose alternative routes [47].

Mathematically, this is a minimization problem whose objective function is nonlinear but convex, in which there are no restrictions regarding the flow in the arcs. Here, the natural restrictions on the capacity of each street/avenue to receive vehicles are implicit, being the result of its own latency function. Further details about this minimization problem and the objective functions can be found in Chapter 8.

In the original model, it is assumed that the time to travel an arc a of the road system depends only on the vehicles flow on a itself. However, when considering arcs whose travel time are influenced by other arcs (for example, non-preferred avenues, where drivers must wait their turn before continuing its journey) this assumption does not reflect the reality. Other factors than the flow on a itself often have a decisive influence on the travel time (these factors are detailed in Section 8.1.1).

[4] Their mathematical model is presented in detail in Section 8.1.

2.2.4 Nesterov & de Palma Model

More recently, Nesterov and de Palma proposed an alternative model [65, 66, 187] to solve the TEP (UE and SO), which has some fundamental characteristics that differentiate it from Beckmann's model:

- Constraints on the arc's capacity are explicitly defined and are not a direct consequence of the latency functions as in Beckmann's model;
- Wardrop's First Principle is the result of loosening complementary restriction conditions of the associated convex optimization problem. Specifically, the model allows latency functions to be discontinuous and unbounded on a finite feasible interval;
- The travel time of each arc is computed using Lagrange Multipliers [10, 14, 44] of the capacity's constraints.

Formally, the model considers a road network $G = (N, A)$, where N is the set of nodes (intersections or zones) and A the set of arcs (streets, avenues, etc.). Each arc $a \in A$ has a *capacity* q_a, i. e., the maximum amount of cars that can travel through it in a given period of time and a *free travel time* $\bar{t}_a$, which represents the time required to travel the arc at the maximum allowed speed.

According to the definition, q_a can never be violated and, while there is available capacity to allocate the traffic agents, there is no reduction in the travel speed of vehicles. Upon reaching the limit of capacity q_a, congestion and delays can then occur. Formally:

Theorem 2.1 *Be (f, t) a traffic assignment. Then (f, t) must meet the following conditions:*

- *The total flow f_a in an arc a should never exceed the arc's capacity: $f_a \leq q_a$;*
- *Below the capacity limit, the travel time t_a of any arc $a \in A$ is equal to its free travel time $\bar{t}_a$. Upon reaching the capacity limit, the travel time can be set to any value greater than or equal to $\bar{t}_a$, i.e.:*

$$\begin{aligned} &if \quad f_a < q_a \quad then\, t_a = \bar{t}_a; \\ &if \quad f_a = q_a \quad then\, t_a \geq \bar{t}_a; \end{aligned}$$

The total travel time is defined as $\sum_{a \in A} f_a \cdot t_a$. As pointed out in Section 2.2.2, under the SO condition, drivers are managed by an all-powerful central intelligence, which always seeks to minimize the total travel time by assigning a specific route to each driver in the network, ignoring the fact that this probably does not meet individual

interests. According to Nesterov and de Palma, computing a traffic assignment under the SO condition (NdP-SO) corresponds to solving the following minimization problem:

$$
\begin{aligned}
(NdP-SO) min_{f,h} \quad & \sum_{a \in A} f_a \cdot \bar{t}_a & \\
subject\ to \quad & f_a = \sum_{k \in OD} h_a^k \leq q_a & \forall\ a \in A \\
& Eh^k = \delta_k & \forall\ k \in OD \\
& h^k \geq 0 & \forall\ k \in OD
\end{aligned}
$$

where

- $OD \subset \mathcal{V} \times \mathcal{V}$ represents the set of fixed origins and destinations for every demand $d \in \mathcal{D}$;
- $d_k > 0$, $k \in OD$ is the number of vehicles moving from origin to destination of k during a certain period of time;
- $h^k \in \mathbb{R}^{|A|}$ represents the flow of the OD pair $k \in OD$ and, hence, $f = \sum_{k \in OD} h^k$.

and

$$
E_{u,a} = \begin{cases} -1 & \text{if } u \text{ is the tail of arc } a, \\ 1 & \text{if } u \text{ is the head of arc } a, \\ 0 & \text{otherwise.} \end{cases}
\quad
\delta_{k,u} = \begin{cases} -d_k & \text{if } u \text{ is the origin of } OD \text{ pair } k, \\ d_k & \text{if } u \text{ is the destination of } OD \text{ pair } k, \\ 0 & \text{otherwise.} \end{cases}
$$

A detailed comparison between Beckmann's et al. and Nesterov & de Palma models is available in [47].

Although Beckmann's model dates back to 1956, is still the most widely used model for the macroscopic traffic assignment [8, 13, 156, 222, 241]. And, since 2003, the UFG group of traffic studies have used it in the macroscopic urban traffic allocation process. For these reasons, Beckmann's model is used as the basis for the simulation employed in this work, and its implementation is presented in detail in Chapter 8.

2.3 General Remarks

The present chapter depicted the basic concepts of urban traffic simulation and the models available for studying the traffic assignment problem. It also provided a more detailed description of the macroscopic model and presented two well known proposals for this kind of traffic assignment.

The concepts presented here and discussed in further detail in Chapter 8 are of fundamental importance for understanding how a macroscopic traffic assignment can be achieved. These concepts ultimatelly guided the parallel algorithmic strategies adopted throughout the current thesis.

CHAPTER 3

Parallel Computing and the GPU

This chapter presents the basic terminology and some important concepts of parallel computing, as well as a detailed description of the technologies behind the new *GPUs* (***G**raphics **P**rocessing **U**nits*). It is organized as follows: Section 3.1 presents some basic concepts of parallel computing. Section 3.2 gives a general vision of modern GPUs, depicts some good GPU programming strategies and tools for GPU programming. Section 3.3 details some advanced techniques used in this work to better explore parallelism. Finally, Section 3.4 gives some general remarks about the main concepts presented in the chapter.

3.1 Background

Parallel computing is a technique in which multiple instructions are simultaneously executed using different processing units, according to the idea that a complex and/or large problem can be "broken" into smaller sub-problems and these, in turn, can be solved simultaneously [221].

Unlike sequential computing, where the *Von Neumann* execution model [243] is the only well-established one, in parallel computing there is no consensus about the best execution model, with a miriad of proposals that have emerged in the last decades. As the study of these models is outside the scope of this thesis, only the best known (PRAM) is presented in more detail in Appendix A. Campbell [37] and Hartenstein [125] present surveys about the most important ones.

Although for a long time there has been a strong skepticism about the real benefits of parallel computing, especially due to Amdahl's law [4], after the 1988's article by Gustafson [118], interest in the subject has risen and, currently, *hardware* and *software* manufacturers spend huge efforts trying to explore this technology. In the frantic race to determine who has the fastest supercomputer, all the more powerful computers around the world make use of thousands of multi-core processors running in parallel and, in some cases, aided by hundreds or thousands of the new, programmable GPUs (see Section 3.2).

Basically there are four different types of parallel computing: bit-level, instruction-level, data and task parallelism. They are briefly described next:

- ***Bit-Level Parallelism*** – Starting in the 70's, with the introduction of VLSI chip manufacturing technology (very large scale integration), advances in computer architectures were achieved by increasing the length of the words that the microprocessor can handle in each clock cycle. By increasing word size (i.e., number of bits that form it) the amount of instructions that the microprocessor needs to perform when processing variables that exceed this size is reduced.
  As a simple example, consider the steps that an 8-bit processor must follow when instructed to add two 16-bit integers. In this case, it first adds the low order 8 bits of the two numbers, then it adds the 8 bits of high order using an "add-with-carry" instruction and the transport bit (carry bit) from the initial addition. Thus, an 8-bit processor must perform two steps to complete the operation, while a 16-bit can complete it in one step.
- ***Instruction Level Parallelism*** – A computer program is nothing more than a stream of instructions executed by a microprocessor. In many situations these instructions can be rearranged and grouped so as to be executed in parallel without affecting the final result of the program. Advances in this area were the focus of the microprocessor industry between the mid-80s and 90s.
  Modern processors have an *instruction pipeline*[1] with several stages. Each stage in the pipeline corresponds to a different action the processor executes in the instruction at that stage. In other words, a processor with N pipeline stages may have up to N instructions in different stages to complete. See Figure 3.1.

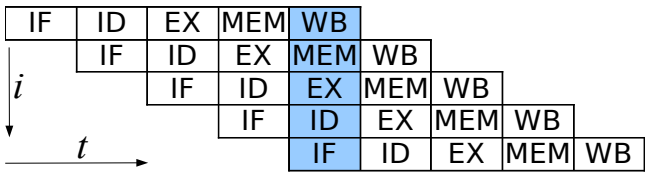


Figure 3.1: *A 5-stage pipeline of a RISC machine. IF = Instruction Fetch, ID = Instruction Decode, EX = Execute, MEM = Memory Access, WB = Register Write Back.*

In addition to the pipeline, some processors are able to execute more than one instruction at a time. They are known as superscalar processors (see Figure 3.2). Instructions can be grouped only if there is no data dependency between them.

[1] ***Pipelining***: hardware design technique where the processor is able to execute more than one instruction at a time, without waiting for a instruction to finish before starting the next one.

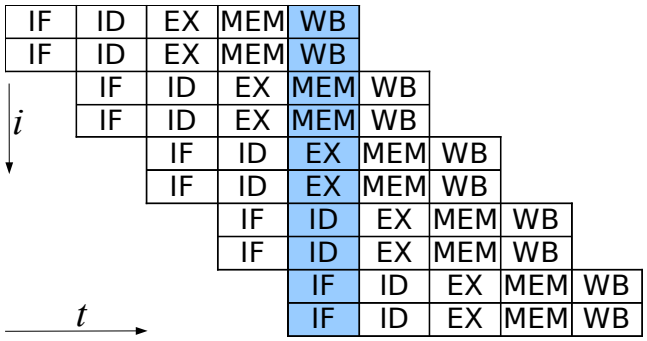


Figure 3.2: *A 5-stage pipeline on a superescalar processor, able to handle two instructions per clock cycle. There can be two instructions on each pipeline stage, with up to 10 instructions (the highlighted column) being simultaneously executed.*

- ***Data-Level Parallelism*** – This type of parallelism explores the concurrence that emerges from the application of a similar (but not necessarily identical) sequence of operations to multiple elements of a data structure [94] and is typically implemented by a repeating loop. For example, "*add 4 to all elements of this vector*" or "*increase by 30% the salary of all employees with five or more years of formal contract*". It is often found in programs that work with large volumes of data.
  However, there are situations that prevent a loop to be parallelized. If the data being processed has values that depend on previous iterations, then the technique cannot be applied. For example, the pseudo-code presented in Algorithm 3.1, which computes the k-th term of Tribonacci sequence [79], can not be parallelized because $value$ depends on n_1, n_2 and n_3 and these are calculated in each iteration of the loop.
- ***task level parallelism*** – Here, different instruction sequences may be executed simultaneously, either on the same or distinct data sets, in contrast to data-parallel level, where a similar calculation is performed on different sets of data. The latest multi-core processors from AMD, ARM and Intel can perform this type of parallelism.

3.1.1 Amdahl's Law

In 1967, the computer architect Gene Amdahl [4] established a law that, for a long time, was extensively used as a way to measure the maximum *speedup*[2] that could

[2]Value that measures how faster a parallel program is, when compared to the corresponding sequential version.

Algorithm 3.1: *TribonacciSequence*()

Input: $K > 3$
Output: k-th term of Tribonacci sequence.

```
1  n1 ← 1
2  n2 ← 1
3  n3 ← 2
4  idx ← 3
5  while idx < K do
6  |   idx ← idx + 1
7  |   value ← n1 + n2 + n3
8  |   n1 ← n2
9  |   n2 ← n3
10 |   n3 ← value
11 end
12 return value
```

be achieved when programs run on parallel processors versus only one processor, since the size of the problem to be solved remains the same when the algorithm is parallelized.

In a synthetic way, the law states that the achievable performance gains that can be obtained when parallelizing a program[3] are limited by its inherently sequential part.

The law can be formalized as follows: let "***s***" be the fraction of the program to be sequentially executed and let "***p = 1-s***" be the parallelizable fraction, with $0 \leq s \leq 1$. The achievable speedup by a parallel computer equipped with "***N***" processors is

$$S = \frac{1}{s + \frac{p}{N}}$$

To illustrate this phenomenon assume that, for a given problem size, 18% of the program must necessarily be executed sequentially (for example, connecting to the database, environment initialization, etc.), while the remaining 82% is dedicated to solve the problem and this part is parallelizable. Amdahl's law says that the speedup that can be achieved when parallelizing on a machine with, let's say, 16 processors is:

$$S = \frac{1}{0.18 + \frac{1-0.18}{16}} \approx 4.32$$

and the maximum achievable speedup would be:

$$\lim_{N \to \infty} \frac{1}{0.18 + \frac{1-0.18}{N}} \approx 5.56$$

[3] Any program, no matter how complex it may be, will always have pieces of code that are inherently sequential.

Due to Amdahl's law, for a long time there was a lot of pessimism about the usefulness of massively parallel computing. Even if the serial fraction "***s***" of the problem is very small and the amount of processors grows indefinitely ($N \to \infty$), the law says that the maximum speedup that can be achieved is only $\frac{1}{s}$. Figure 3.3 illustrates the phenomenon for $N = 1024$: The speedup drops dramatically even for very small serial fractions [118].

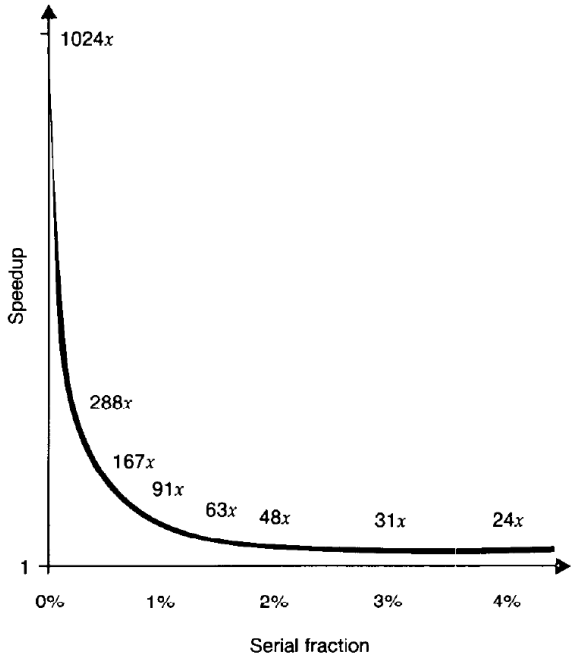


Figure 3.3: *Maximum speedup under Amdahl's law (extracted from [118]).*

3.1.2 Gustafson's Law

In 1988, John L. Gustafson published an article [118] questioning the validity of Amdahl's law. The author argues that there is a flaw on it: the assumption that "***p***" is independent of "***N***", which *practically never happens*. It is unusual to get a fixed-size problem and try to solve it with a certain variable amount of processors, except perhaps in the academic world. In practice, *the problem size grows along with the number of processors*. Given more computational power, the problem usually grows to make use of such power. Consequently, it is more realistic to assume that the *runtime* is constant, not the *problem size*.

He also states in his article that the parallelizable is the part of the program that grows, following the size of the problem. Program tasks such as startup, bottlenecks in communication, among others composing "***s***" *do not grow* (or grow slowly) with the problem. This means that, in a first analysis, the amount of work that can be performed in parallel *varies linearly with the number of processors*.

In a parallel system with "N" processors, using respectively "s_{er}" and "p_{ar}" to represent the time spent in serial and parallel parts, then a single processor will require a $s_{er} + p_{ar} \times N$ time to complete the task. Assuming that $s_{er} + p_{ar} = 1$ for algebraic simplicity, this leads to an alternative to Amdahl's law suggested by E. Barsis [118]:

$$\begin{aligned} Scaled\ speedup &= \frac{(s_{er} + p_{ar} \times N)}{s_{er} + p_{ar}} \\ &= (s_{er} + p_{ar} \times N) \\ &= (s_{er} + (1 - s_{er}) \times N) \\ &= N + s_{er} - N \times s_{er} \\ &= N + (1 - N) \times s_{er} \end{aligned}$$

As it can be seen in the equation, it is much easier to get a significant performance increase with parallelism than Amdahl's law suggests. Figures 3.4 and 3.5 illustrate and summarize these two arguments.

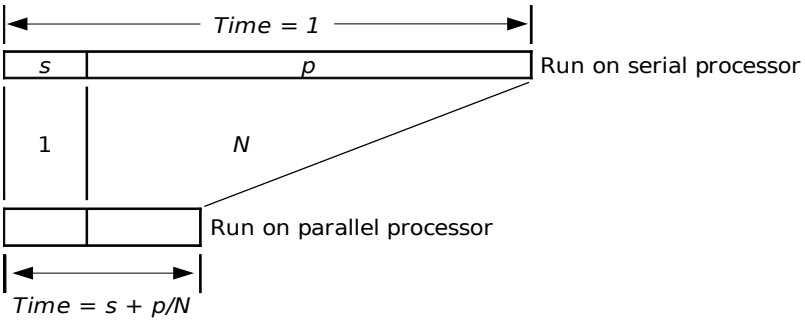


Figure 3.4: *Fixed size model for Speedup* = $\frac{1}{s+\frac{p}{N}}$ *(extracted from [118]).*

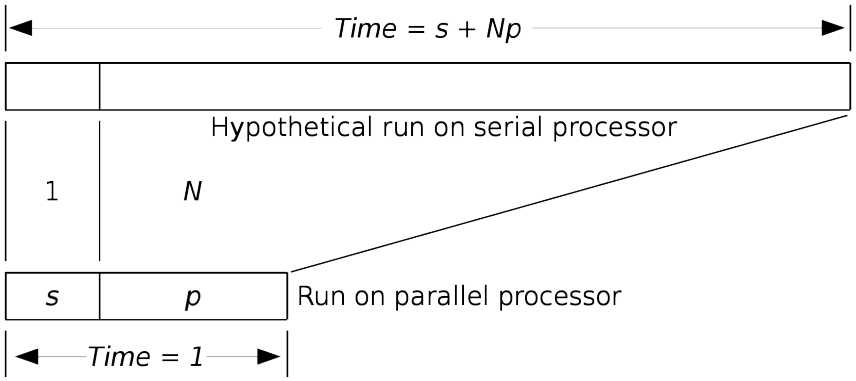


Figure 3.5: *Scaled size model for Speedup* = $s + N \cdot p$ *(extracted from [118]).*

3.1.3 Flynn's Taxonomy

In 1966, Flynn et al. [89, 195, 204] proposed a classification for computer systems – which ended up becoming the standard and is still widely used to this day – based on the idea of *instruction flows* and *data flows* to be simultaneously processed. This classification divides computers into four classes considering the number of instruction streams (single or multiple) and data streams (single or multiple), categorized as:

- ***S**ingle-**I**nstruction **S**ingle-Data **S**treams* (SISD): *von Neumann* machines, purely sequential, are classified in this category;
- ***S**ingle-**I**nstruction **M**ultiple-Data **S**treams* (SIMD): defines parallel machines that have a single control unit and where all processors execute the same instruction on different data sets synchronously and in lock-step;
- ***M**ultiple-**I**nstruction **S**ingle-Data **S**treams* (MISD): in this category, the same set of data is handled by processors executing different instruction streams. In practice there are no viable MISD machines. However, some authors consider that there are machines that can be classified as MISD [80];
- ***M**ultiple-**I**nstruction **M**ultiple-Data **S**treams* (MIMD): unlike SIMD machines, MIMD have a control unit for each processor and are able, therefore, to execute different instructions at different datasets.

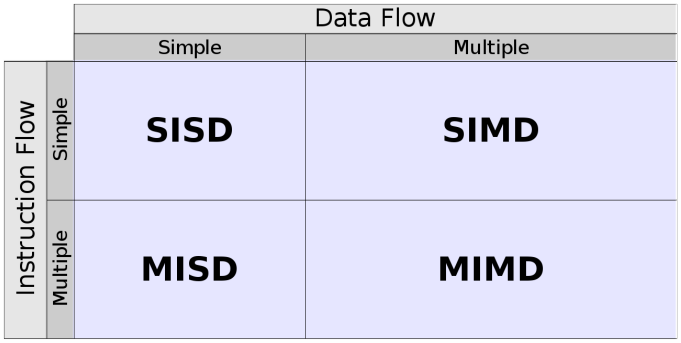


Figure 3.6: *Flynn's Taxonomy for computer systems.*

In 1988, Johnson [140] proposed a division in the classification of MIMD machines, based on memory structures (global or distributed) and the communication and synchronization mechanisms (shared variables or message passing), as shown in Figure 3.7.

According to the memory structure and the ways of communication and synchronization, the MIMD architecture can be divided in:

- ***G**lobal **M**emory **S**hared **V**ariable* (GMSV): multiprocessor systems with shared memory and considered tightly coupled;

- ***D**istributed **M**emory **S**hared **V**ariable* (DMSV): implement distributed memory combined with programming through shared variables. Also known as systems with distributed shared memory;
- ***D**istributed **M**emory **M**essage **P**assing* (DMMP): multicomputer systems with distributed memory and considered loosely coupled;
- ***G**lobal **M**emory **M**essage **P**assing* (GMMP): machines with shared global memory and where the processors communicate through message passing. They are little used in practice [195].

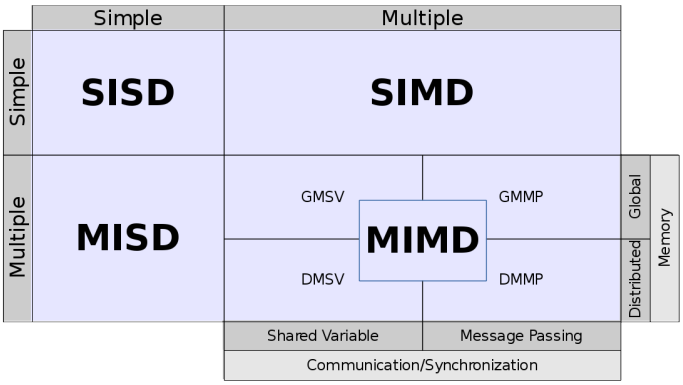


Figure 3.7: *Flynn-Johnson's Taxonomy for MIMD machines.*

Next, more details about the SIMD class are presented, as it is related to the hardware architecture that is the focus of this thesis.

3.1.4 SIMD Machines

Processing units in parallel computers can operate both under the supervision of a central controller and in an independent manner [115]. Machines built under *SIMD* philosophy fit into the first category. This central controller, in turn, is responsible for retrieving and interpreting the program instructions, whether sequential or parallel. It is the task of the central controller to identify data structures and processing instructions that can be parallelized and to transfer them to the processor set (see Figure 3.8). When identifying any control flow or calculation that can not be parallelized, the central controller handles the data to be processed.

The history of SIMD machines dates back to 1962, with the ILLIAC IV [61, 131, 170] project. It was the first large scale multi-processed machine, consisting of 64 processing units. The project ended up abandoned due to high costs and low performance, taking with it the SIMD concept. It was only in 1985 that Dannis Hillis ressurected the SIMD architecture with his "Connection Machine" [130], consisting of 65,536 1-bit processors [144]. More recently, SIMD concept variants found fertile ground in co-processing units like MMX of Intel/AMD processors, in digital signal processing

chips (DSP) and on SSE technology (Streaming SIMD Extensions) also implemented on Intel/AMD processors [115]. The SSE provides a set of instructions that perform the same operation in one or more sets of data.

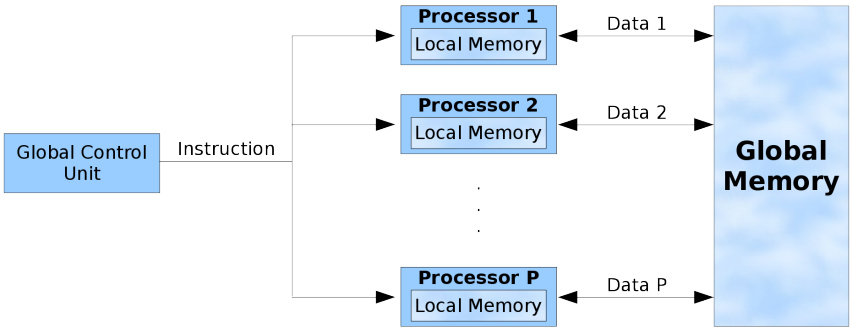


Figure 3.8: *Processing Flow on a SIMD Machine.*

Current GPUs are not strictly SIMD machines, but operate in a pretty similar way. They actually have strengthened the SIMD concept with the ability of multithreading, creating a new execution model called *Simple Instruction, Multiple Threads – SIMT*.

In this model, a set of "p" processing elements seems to execute much more than "p" tasks. This can be accomplished by letting "p" to run multiple "work-items" (or "threads") which, by its turn, execute in a lock-step analogously to SIMD [172].

The SIMT execution model was firstly introduced by NVidia in the G80 GPU [164]. AMD released the R600 GPU chip under the same philosophy a short time later [132].

GPUs also operate under the *Simple Program, Multiple Data – SPMD* concept. Here, each GPU's processor executes the same program, but every processor can possibly handle a different amount of data or take slightly different paths during program execution, breaking the lock-step rigid imposition of the SIMD model.

3.2 A General Overview on Modern GPUs

The ***G**raphics **P**rocessing **U**nits* are micro-processors dedicated to perform operations related to 2D and 3D graphics applications. Among these applications, CAD, CAM, games and graphical user interface can be cited. Thanks to their highly parallelized and specialized architecture, they are much more efficient in handling graphics than general purpose CPUs, which are especially designed for the execution of sequential code.

In essence, a GPU consists of several units specialized in performing floating point operations, massively used in the graphic functions of rendering algorithms. The large ammount of these execution units, working in parallel, is what allows the high computational power of such processors [42, 55].

Currently, GPUs can be used not only to process graphic data but also as math co-processors, running algorithms traditionally managed by CPU, the so called "*General-Purpose Programming on Graphics Processing Units (GPGPU)*". Traditionally they are found in an independent device called *video board* which communicates with the central processing unit through the PCI Express bus (see Figure 3.9 for a simplified representation) [252]. However, since the launch of the AMD Accelerated Processing Units (APUs), NVidia Tegra[4] and Iris Graphics (Intel), GPUs can also be found as an integrant part of the CPU. In the latter case, CPU and GPU share the same memory space.

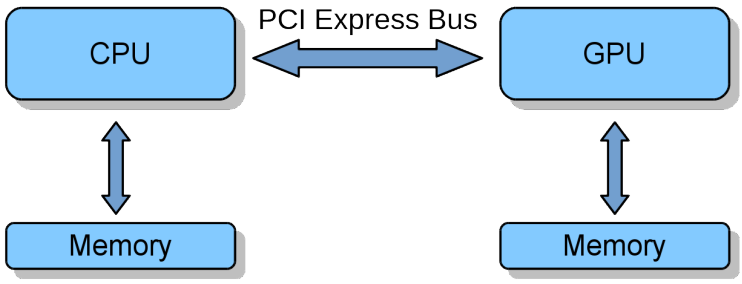


Figure 3.9: *Simplified representation of CPU and GPU communication scheme.*

When CPU and GPU are two autonomous devices, processing data and code to be executed need both to be sent from CPU (the host) to GPU (the device) through the PCI Express Bus. Once the computation is completed, the resulting data is sent back to the CPU's main memory, as depicted in Figure 3.10.

It is worth remembering that CPUs and GPUs have evolved independently. While the first was developed to answer the requests as fast as possible – i.e., to minimize latency – the second have as ancestors 3D graphics accelerators and whose main goal was the maximization of delivered data per time unit. There is not a common ancestor in their family trees or a missing link that unites them [126].

Since the goal of a CPU is to work with the lowest latency possible, it has in its internal design several optimizations aimed exclusively at minimizing the response time to any requests, such as [90]: large amount of space dedicated to the control unit instead of ALUs in order to refine reorder execution, provide instruction parallelism and minimize interruptions in pipeline; multiple cache levels to cover latency; and branch prediction for conditional jumps.

GPUs, on the other hand, try to deliver the largest possible amount of data per time unit – i.e., to maximize the throughput. To achieve this, in their project more transistors are devoted to data processing rather than data caching and flow control [98]. Figure 3.11 depicts the differences between the two philosophies.

[4]Processors based on ARM's architecture, such as the SoCs (system-on-a-chip) used in smartphones, tablets and PDAs.

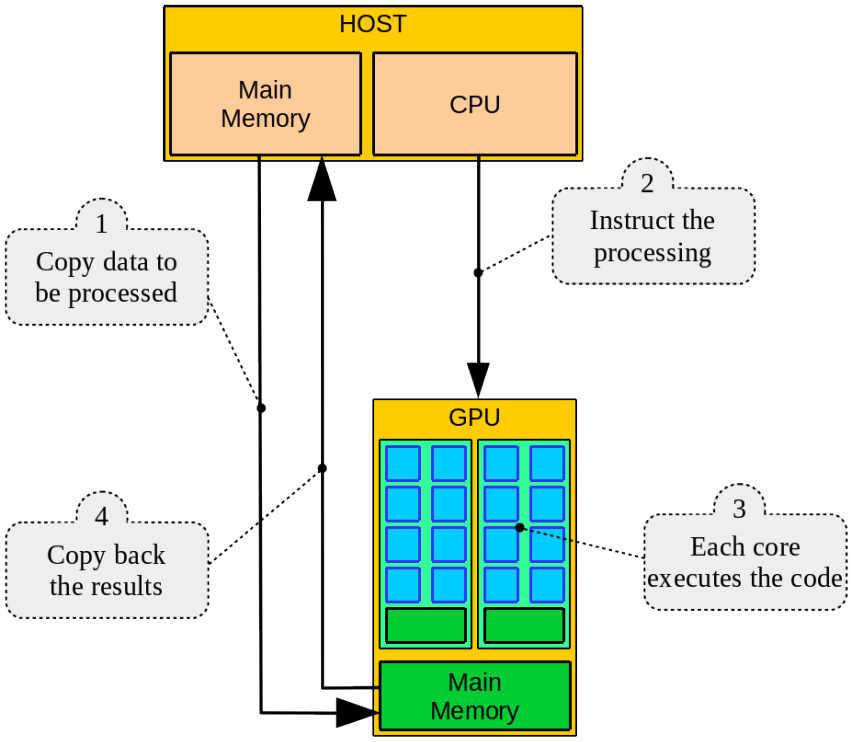


Figure 3.10: *GPU Processing Flow.*

From a high level point of view, modern GPUs are conceived based on three key features [180, p. 11–14]:

1. ***Stream Processor – SP***. Also known as *CUDA core* in NVidia's nomenclature[5] and *Processing Elements – PE*, they are execution – or computing – units able to run algorithms with high efficiency and in synchrony with other SPs.
   The programs running on SPs are called *kernels* and a *work-item* is one of a collection of instances of a kernel, being executed by one or more SPs. Each work-item has identifiers able to distinguish it uniquely within the group of work-items in a SM (local identifier) and globally among all work-items (global identifier).
   The two main parts of a SP are the *Arithmetic Logic Unit – ALU* and the *Floating Point Unit – FPU*. Each SP has access to a small, but extremely fast, memory for its private use;
2. ***Streaming Multiprocessor – SM***. SPs do not operate independently. They are grouped in tightly coupled multiprocessor blocks called *Streaming Multiprocessors*. SMs not only group the SPs, they also provide a way of communication through a shared memory mechanism. The work-items executed by the SPs are grouped in *wave-fronts* and the work-items inside a *wave-front* execute the programs in a SIMD style; otherwise they work in a SPMD fashion. In the present thesis, we

[5] See Table 3.1 for an equivalence list between OpenCL and CUDA terms.

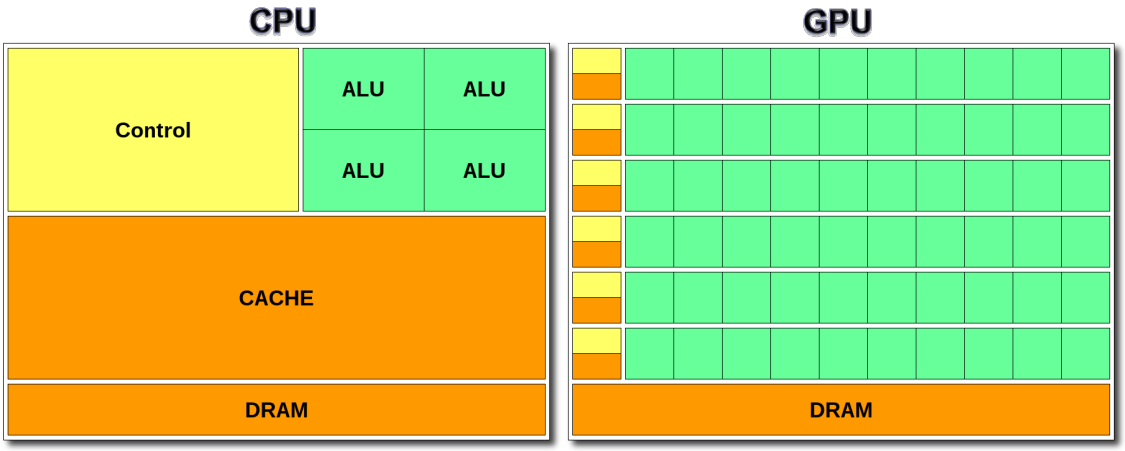


Figure 3.11: *Differences between CPU and GPU internal architectures.*

define *MaxSMSize* as the maximum number of work-items every SM can handle simultaneously. See Figure 3.12;

3. ***Memory Hierarchy***. According to OpenCL specifications there are four types of memory in a GPU [238]. They are briefly described next:

   - ***Global*** – Available to all SPs for read/write. It is the only way processors grouped in different SMs can communicate and, in advanced models, it can reach the amount of tens of gigabytes. It is the device's main memory and, despite the large space available, it is considered slow when compared to memories in other hierarchies;
   - ***Constant*** – As well as the global memory, this kind of memory is available to all SPs and is physically located in the device's main memory[6]. However, it can be used more efficiently than the first one if the executing units are equipped with hardware that supports constant memory cache;
   - ***Local*** – Can be read or written by all SPs inside a SM. It's usually available at a much smaller amount than the global memory, ranging in each SM from 16KB in the most basic models up to 64KB in high-end models, being the second fastest in GPU memory hierarchy;
   - ***Private*** – Memory that can be used only within each SP during the execution of the program; they correspond to GPU's registers and, usually, have an access time 6 to 10 times faster than local memory.

The Figures 3.12 and 3.13 give a general overview on how all this elements are combined to form a graphic processing unit.

[6] Although, on the CPU side, this kind of memory can be read/write, on a GPU it is always read-only, cached and with its size usually around 64kb per SM. When compared to global memory, it is somewhat closer to the processor and much faster to access. It is, however, slightly slower than local memory.

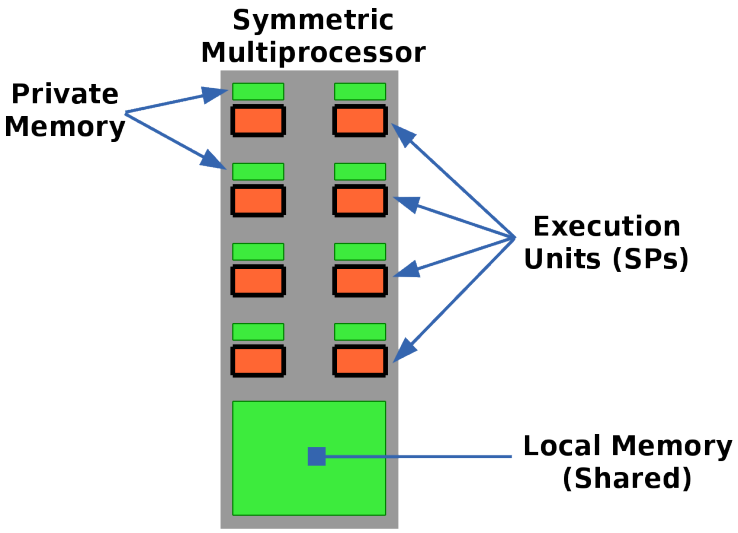


Figure 3.12: *High level view of a Streaming Multiprocessor – SM.*

From the point of view of the internal organization, each *lane* of *hardware* of each SIMD *Streaming Multiprocessor – SM* (in [126], Hennessy and Patterson present further details about the internal organization of GPUs) is virtualized in large batches of *work-items*, called *wave-fronts*. Each wave-front is composed of 64 *work-items* (32 threads in the *CUDA warps*) and this number, in turn, is a multiple of the length of each *lane*.

All work-items within a wave-front operate in SIMD model, executing the same instruction coordinated by a central clock. Multiple wave-fronts, in turn, are combined into larger structures, called *work-groups*, where the running work-items can communicate and share data through a local (shared) memory. Finally, multiple work-groups can be queued via hardware to run on each SM.

It is worth a short comment here about the way SPs operate in AMD GPUs based on HD2000 architectures and later ones. Since then, each functional unit is arranged as a 5-way superscalar shader processor. That means that each SP is able to manage 5 scalar floating point instructions per clock cycle and one of them (the fat one in Figure 3.14) can handle transcendental[7] functions.

3.2.1 Good GPU Programming Strategies

Based on the way modern GPUs are organized, some strategies are essential to extract the maximum performance of the programs running on them:

[7] A function that does not satisfy a polinomial equation whose coefficients are themselves polynomials. Trigonometric, exponential and logarithmic are examples of that kind of functions.

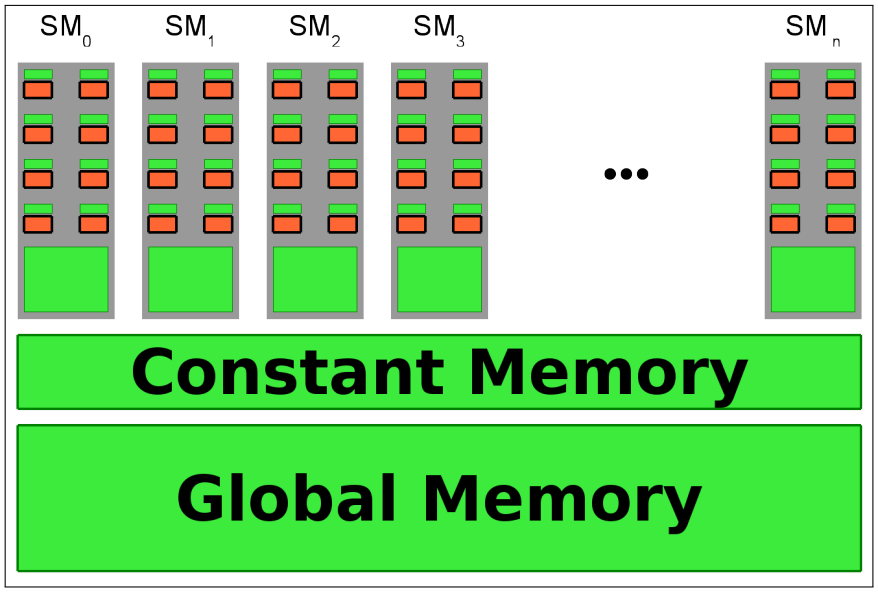


Figure 3.13: *High level vision of a GPU architecture.*

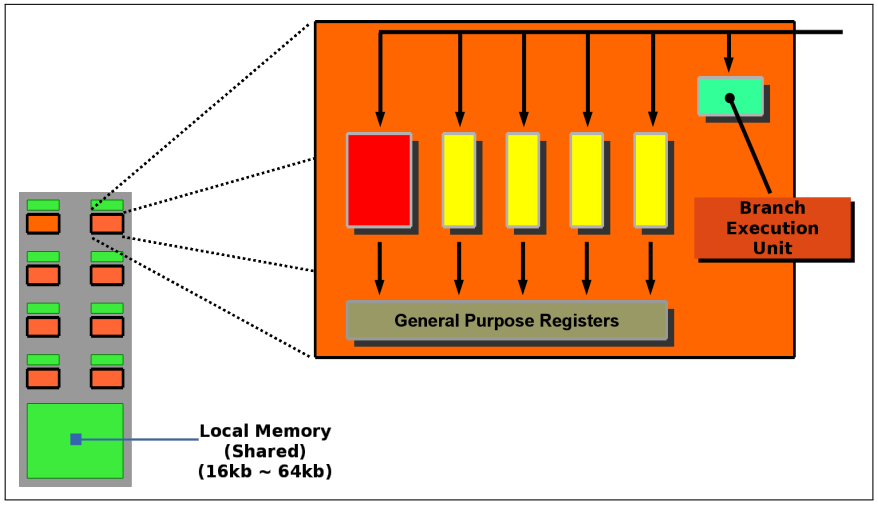


Figure 3.14: *AMD Stream Processor.*

- The use of the local memory should be maximized since, as stated before, it is the second fastest in GPUs' memory hierarchy. It was designed in order to allow high speed access in a massively parallel way. Despite the small size, its efficient use is one of the key aspects for the implementation of fast programs on these devices;
- Design the algorithms in order to avoid bank conflicts when writing to local memory. In the GPU, the memory circuits of the local memory are arranged in the form of banks of the same size that can be accessed simultaneously in a single transaction. Each bank is organized in groups of 32 bytes (integers or single precision floats) in a way that each consecutive word is stored on a different bank. When a wave-front carries out a write transaction in this memory and the writing operation is performed in different banks, the transaction can be carried out in parallel. In contrast, when work-items in the same wave-front try to perform a

Table 3.1: *Equivalence between OpenCL and CUDA terms*

OpenCL	CUDA
Device	GPU
Streaming Multiprocessor (SM)	Compute Unit (CU)
Processing Element (PE)	Scalar Core
Stream Processor (SP)	CUDA Core
Global Memory	Global Memory
Local Memory (per SM)	Shared Memory (per CU)
Private Memory	Local Memory
kernel	kernel
work-group	block
work-item	thread

writing access in the same bank, the operation cannot be done in parallel and the GPU serializes the process, leading to a performance bottleneck known as *shared memory bank conflict*;

- The use of the global memory must be minimized. Although being available in large amounts, it has an access time a few orders of magnitude higher than local memory, and its constant use (read and/or write) will severely degrade the application's performance;
- If access to global memory is necessary, such access should be performed in a coalesced form. A *coalesced memory access* or *memory coalescing* is the technique of combining multiple memory accesses into a single transaction, speeding up the read/write operations. For example, if a group of 64 work-items (a wave-front) needs to access a sequence of 256 successive bytes (64 integers or single precision floats) this can be done by the hardware in one single transaction. On the other hand any non-sequential, sparse or misaligned access to global memory will potentially downgrade the speedup. In short: consecutive work-items should always try to access consecutive memory locations;
- Work-items should be kept busy as much as possible doing some useful processing. Reading data from global memory, performing a simple operation and then writing the result back to global memory is not a smart way to use the GPU resources. See Section 3.3.2 for further details;
- Communication between CPU and GPU should be avoided to the maximum. Since they are distinct devices, all communication among them must be performed through the PCI Express bus, a much slower medium than the ones available within the GPU device. Ideally, all necessary data should be sent to the GPU only at the beginning of the computation and read by the CPU at the end;
- The use of conditional branches (like "else", "switch-case", etc) should be mini-

mized. Work-items inside a wave-front must follow the same execution flow, running exactly the same instruction at the same time (i.e., under the SIMD model). During execution of a program, if a conditional statement like an "if-then-else" is reached, the GPU will have to run the "true" part (then) first, deactivating all work-items that go to the "false" part (else) and invert the situation after finishing the "true" part. As it will be discussed later, in Section 3.3.3, when this situation happens, the two branches of the conditional statements are not executed in parallel, but in a serial fashion, which can result in a significant and undesirable performance loss.

3.2.2 Tools for GPU Programming

The purpose of this section is to describe the current state of the various programming tools for GPUs, as well to briefly depict some of their capabilities.

- ***DirectCompute*** – Part of the DirectX collection of APIs since version 10, it was released by Microsoft in late 2009 [177]. Despite being hardware independent, it is designed specifically for machines running Windows (Vista and newer releases), which ultimately limits the scope of software developed under the platform.
- ***Compute Unified Device Architecture (CUDA)*** – It is NVIDIA's parallel computing architecture that allows the exploration of the power of GeForce, ION, Quadro, Tegra and Tesla GPU(s) for general purpose computing, as well as games. Using it, software developers, scientists and researchers can process video and images, computational biology and chemistry, fluid dynamics simulation, reconstruction of CT images, seismic analysis and ray tracing, among other applications.
  Programs to be executed by NVidia GPUs must be written in "C for CUDA" (regular C/C++ code with NVIDIA extensions and certain restrictions) and compiled by the NVidia compiler (nvcc). Software tools like Mathematica and MatLab provide native support for CUDA, while third-party bindings allow the use of CUDA in Python, Perl, Fortran, Java, Ruby, Lua, MATLAB and IDL. Currently, CUDA also allows code to be written in OpenCL.
- **APARAPI –** ***A Parallel API*** – It is an API launched by AMD in October 2010 and released under the GPL in September 2011. It can be used to make compatible programs – written in Java – to execute directly on AMD GPUs, without having to re-write the code in OpenCL or CUDA. Without any user intervention, APARAPI analyzes the program sources and checks the availability of a compatible GPU. If any video device is available, the data will be transparently sent to such a device using OpenCL. If a compatible GPU is not available, the API automatically tries to run the code in multiple CPUs, in order to get the maximum available performance.

Java developers can now program the GPU without knowing CUDA or OpenCL, thus avoiding steep learning curves and new training in GPGPU languages.

- ***Open Computing Language (OpenCL)*** – It is an open standard originally defined by Apple and now maintained by the Khronos Group, for generic use of parallel computing in heterogeneous environments, such as CPUs, GPUs, CELL Broadband Engines, NPUs (Network Processing Unit), etc. It comprises an API (used to manage OpenCL entities, such as kernels, environments, computing devices, and others) and a language based on C, for writing *kernels* (programs that run on *entities*), offering the same features of CUDA and DirectCompute.
  An OpenCL platform (host) is composed of one or more computing devices, such as a GPU. Each computing device comprises computing units which, in turn, consists of processing elements.

The present thesis adopts the Khronos Group OpenCL terminology and programming model throughout the text. Table 3.1 presents an equivalence list between OpenCL and CUDA terms.

3.3 Advanced Parallel Techniques

This section details some advanced techniques to further explore parallelism and that were extensively used in the present work.

3.3.1 Loop Unrolling

Loop Unrolling (also known as *Loop Unwinding* and *Loop Unfolding*) is an optimization technique – performed by the compiler or manually by the programmer – applicable to certain kinds of loops in order to reduce (or even prevent) the occurrence of execution branches and minimize the cost of instructions for controlling the *loop* [1, 91, 133, 219]. Its goal is to optimize the program's execution speed at the expense of increasing the size of the generated code (*space-time tradeoff*). It is easily applicable to loops where the number of executions is previously known, like routines of vector manipulation where the number of elements is fixed.

Basically the technique consists in the reuse of the sequence of instructions being executed within the loop, so as to include more of an iteration of the code every time the *loop* is repeated, reducing the amount of these repetitions.

This reuse is done by manually replicating the code inside the *loop* a certain amount of times or through the "*#pragma unroll* ***n***"[8] positioned immediately before the

[8] A *directive pragma* is a language construct that provides additional information to the compiler,

beginning of the loop. The number of times the loop is unrolled is called *Unrolling Factor* and, with the pragma directive, it is given by the parameter "***n***".

It is worth noting that with the pragma directive we leave the decisions of how the loop should be unrolled to the compiler, which may lead to a not so optimized resulting code. In the experiments performed as part of this thesis, the best results were always achieved using manual loop unrolling, reason why this strategy has been chosen in the current work.

As an example, consider the C code shown in Listing 3.1, which simply multiplies the elements of an array by its index ($a_i \leftarrow a_i \cdot i$). In this example, we call ***L*** the *loop size* and ***F*** its *unrolling factor*. ***L*** here is equal to 100.

Listing 3.1: *Multiplying elements in a vector*

```
for (int i = 0; i < 100; i++) {
 a[i] = a[i] * i;
}
```

It's possible to significantly improve the execution speed of this algorithm by unrolling it, as shown in Listing 3.2.

Listing 3.2: *Unrolling the multiply routine*

```
for (int i = 0; i < 100; i += 3) {
 a[i]   = a[i]*i;
 a[i+1] = a[i+1]*(i+1);
 a[i+2] = a[i+2]*(i+2);
}
```

The two extra lines of code and the "*i += 3*" in Listing 3.2 performs the desired three-fold ($F = 3$) manual loop unrolling.

As it can be seen, the $\frac{L}{F}$ ratio does not necessarily need to be an integer. If it admits a remainder, the compiler can (since the number of iterations is previously known at compile time) add extra code to the end of the unrolled generated code in order to ensure its correctness.

Unrolling, when applicable, offers several advantages over non-unrolled code. Besides the decrease in the number of iterations, an increase occurs in the amount of work done each time through the loop. This also open ways for the exploration of parallelism by the compiler in machines with multiple execution units, since each instruction within the *loop* can be handled by an independent thread.

specifying how to process its input. This additional information usually is beyond what is conveyed in the language itself.

However, these are only the most easily perceivable benefits. Agner Fog [91] listed several others, as well as some observations about when this technique should be avoided. Such factors (advantages and disadvantages) must be considered by the programmer when deciding to use loop unrolling or not.

3.3.2 Persistent Threads

Since the launch of the first programmable GPUs and with all its basic architecture inspired by the SIMD model, the "*Single Instruction Multiple Thread"* (SIMT) and "*Single Program Multiple Data*" (SPMD) paradigms have become standards *de facto*. Both seek to hide the details of the underlying *hardware* where the code runs, attempting to facilitate the painful task of development [117].

Gupta et al. [117] argue that the usage of these "traditional" paradigms greatly limits the actions of the programmer, because all control of the execution flow is in the power of the *scheduler's* video card. This programming style, which delegates all the decisions to the scheduler, is called by the authors as "non-PT", or "non-Persistent".

It requires that the software developer abstracts units of work to virtual work-items. Since the number of wave-fronts to create is based on the number of virtual work-items, during a kernel launch usually there are several hundreds of even thousands more wave-fronts to be executed than the amount of physical processing elements to assign them to.

Such scheduling of wave-fronts is performed by the *scheduler* and the programmer has no means to interfere in the process, e.g., *how*, *where*, *when* and in *which order* the work-groups will be assigned.

Gupta et al. claim that, while these abstractions reduce the effort for new developers in the GPGPU field, they also create obstacles for experienced programmers, who normally face problems for which workload is inherently irregular, therefore making it much more difficult to efficiently parallelize when compared to problems whose parallel solution is more regular.

According to Gupta et al., this uncovers a serious drawback of the current SPMD programming style, which is not able to ensure *order*, *location* and *timing*. It also does not allow the software developer to regulate these three parameters without completely avoiding the GPU scheduler.

Thus, to overcome these limitations, developers have been using a programming style called *Persistent Threads ("PT")*, whose low level of abstraction allows performance gains by directly controlling the scheduling of work-groups. And although this style has been in use for some time, only in 2012 it was formally introduced, described

and analyzed by Gupta et al. [117]. They also list several problems when adopting the traditional style.

Basically, what the PT style change is the *lifetime* of a *work-item* [184], by letting it keep running longer and giving it much more work than in the traditional "non-PT" style [230]. This is done circumscribing the logic kernel (or part of it) in a loop, so this loop remains running while there are items to be processed.

Briefly, from the point of view of the developer, all work-items are active while the kernel is running. As a direct consequence of PT, a *kernel* should be triggered using only the amount of *work-items* that can be executed concurrently by each Streaming Multiprocessor. All these actions will prevent constant return of control to the host and the cost of new kernel invocations [184].

Gupta et al. acknowledge, however, that the technique of Persistent Threads is not a panacea, and its use should be carefully evaluated [117]. In particular, the technique fits well when the amount of memory accesses is limited (i.e., few reading/writing to global memory and a large volume of computation) and the problem being solved has not many initial input elements or the growth in the number of elements in the input set is fairly limited. Beyond these conditions, the traditional non-PT style tends to outperform the PT style.

3.3.3 Thread Divergence

Current GPUs are able to deliver massive computational power at a reasonably low cost. However, due to the way they are constructed (see Section 3.2), some obstacles must be overcome for the effective use of such power. One of the main and hardest obstacles to avoid is the presence of conditional statements [256] potentially leading to branches in the execution flow of the various work-items [122].

By default, GPUs try to run all the work-items inside the wave-fronts in the SIMD model. However, if the code being executed has conditional statements that lead to divergences in program flow, the divergent work-items will be stalled and its execution will only happen after the non-stalled work-items have completed their runs, which ultimately compromises the desired *speedup*. This phenomenon is called *Thread Divergence* [40, 122, 183, 256].

The two program excerpts presented in Listing 3.3 and Listing 3.4 (adapted from [122]) are examples of this phenomenon.

Listing 3.3: *First example of divergent condicional "if-then-else"*

```
int tid = get_local_id(0);
if (a[tid] > y) {
 ++x;
} else {
 --x;
}
```

Listing 3.4: *Second example of divergent condicional "if-then-else"*

```
int smallestValue(int a, int b)
if (a < b) {
 return a;
} else {
 return b;
}
```

Analyzing the code available in Listing 3.3, if the condition becomes true **for at least one** of the *work-items* and the code "++x" is triggered, then ***all*** the *work-items* within a *wave-front* must pass "++x", regardless of the fact that this code is actually executed or not. If the average probability of the condition "if" being evaluated as true is relatively low, this will result in a very poor use of the *wave-front* elements, since most of them will not do useful work. Even in the average case, 50% of *work-items* will be "idle" during this phase of the program, resulting in GPU resource waste [122]. Something similar happens with the code presented in 3.4.

Now consider the routine presented in Listing 3.5.

Listing 3.5: *Third example of divergence: variable size loop*

```
int n = get_global_id(0);
for (int i = 0; i < n; i++) {
 //Do something;
}
```

In this case, the "for" loop will be executed "n" times and the value of "n", in turn, is dependent on the global identifier of each *work-item*. As such identifiers can potentially take very high values, the *work-items* with a small "n" should wait for those with larger workloads, negatively impacting the performance of the whole algorithm.

All this happens because each *work-item* in a *wave-front* needs to be executed in SIMD model and, if this is not possible, the only thing the GPU can do is to serialize the

entire process and, in the case of code 3.5, the faster *work-items* must wait for the slower ones before continuing their own execution.

Trying to circumvent this problem, some strategies have been proposed in order to minimize or even eliminate the effects of such phenomena. Among them, we cite [40, 98, 122, 173, 183, 256].

This phenomena emerged during the implementation of some of the algorithms presented in the current study, such as the evaluation of the arc cost (Section 8.1.1) or during the enumeration of chordless cycles (Section 5.3). Wherefore it became necessary to develop a method to prevent flow divergence, which could ultimately compromise the performance of such a step of computation. The method is detailed at the end of Section 6.3.

3.4 General Remarks

The concepts presented in this chapter about parallel computing and how GPUs operate are of fundamental importance for understanding the algorithmic choices made throughout the entire work.

In particular, Section 3.2 presented and detailed the internal organization of GPUs, their main elements and their memory hierarchy, emphasizing that the intelligent use of this hierarchy is one of the key aspects of an efficient use of GPUs architectures.

Still in the line of a good use of GPU resources, strategies consolidated in the literature were presented, as well as some advanced parallel techniques.

CHAPTER 4

Parallelism and the Traffic Assignment Problem

As noted in Chapter 1, urban traffic simulators are computationally very demanding, especially if the choice falls on micro or mesoscopic models. This phenomenon, however, is also observed when using macroscopic models with large road networks. For all these cases, the simulations can benefit from the use of parallel computing, which allows to achieve a level of performance and precision – in terms of execution time, size of the networks and amount of traffic details being simulated – not reachable by traditional sequential computing.

This chapter describes some traffic simulation systems implemented using parallel approaches in all their nuances, that is, using multiprocessor hardware, distributed in computer clusters and the latest, making use of GPUs. Since both urban traffic and parallel computing are very broad areas of study, the systems mentioned here only focus on some of their aspects. They were divided according to the adopted simulation philosophy in terms of level of detail (microscopic, mesoscopic and macroscopic) for a better classification.

For a review of some basic concepts related to the study of urban traffic, please refer to Chapter 2.

4.1 Microscopic and Mesoscopic Simulations

This section presents some works describing the use of parallel computing in traffic simulation systems using microscopic and mesoscopic models.

4.1.1 Distributed Simulation

Dai, Zhang and Zhang [59] present a *framework*[1] for distributed microscopic and mesoscopic simulations, called PMTS – *Parallel Microscopic Traffic Simulation*.

[1] Software abstraction layer, with the aim of uniting common code between various development projects, providing generic resources for applications.

Using this framework they built a system for distributed processing using a local high-speed network of computer nodes. The basic idea of parallelism is to divide the simulation of a large urban road mesh in small sub-meshes, each one assigned to a node of the computational network. In the example presented in the article, the simulation area is longitudinally divided into four interconnected portions and, therefore, feasible to be processed in clusters of two or four machines. Figure 4.1 illustrates the partitioning process.

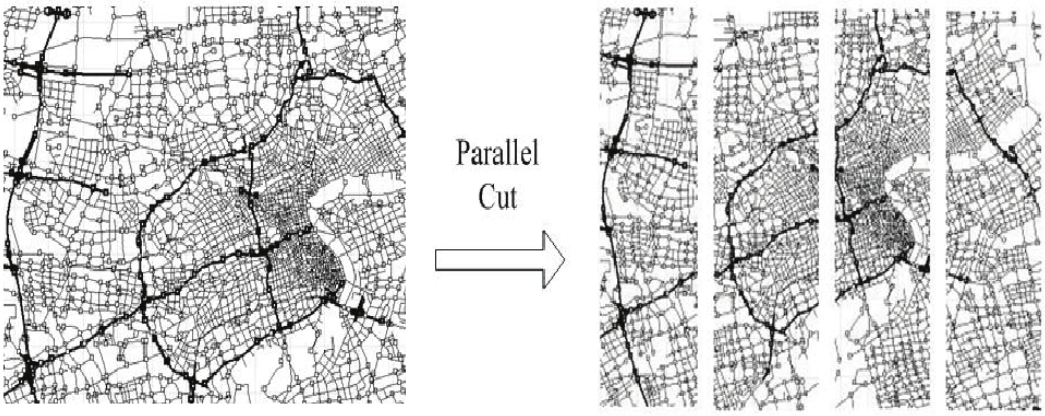


Figure 4.1: *Longitudinal parallel cut of an urban road network (extracted from [59]).*

Once partitioned, each sub-network is sent to the responsible node, which takes care of all aspects of simulation, from the behavior of vehicles to the changes in traffic lights, among others. At each processing node, the simulation is performed based on a three layer software architecture: simulation management, object oriented description and data exchange layer. Figure 4.2 depicts the architecture.

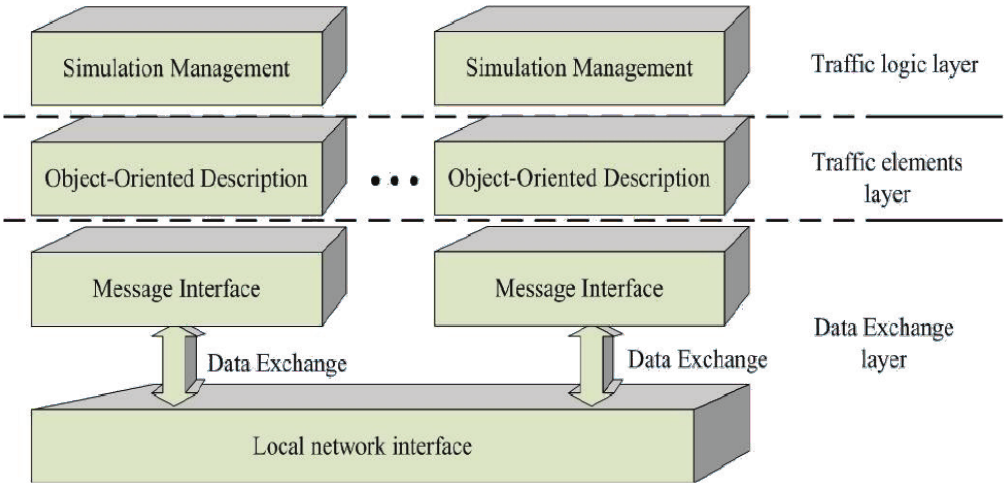


Figure 4.2: *Parallel Microscopic Traffic Simulation Architecture (extracted from [59]).*

The management layer handles all the logic of simulation and also manages user commands such as pause, resume, export the status of the simulation, etc.

In the description layer, all components of urban traffic – cars, avenues, traffic lights, intersections, etc – are specified using object-orientation.

The data exchange layer sends and receives data, synchronizing the nodes of the distributed system, using a (***M**essage **P**assing **I**nterface* – MPI) standard. During the simulation, when a vehicle is about to cross the border between two sub-meshes, MPI sends the data of such a vehicle to the processing node responsible for the destination sub-mesh.

In the tests carried out to evaluate the system's performance, the simulation environment consisted of a cluster of 6 computers, each one with a 2.6 GHz single-core CPU and 1GB of RAM memory, connected via a 100 Mbits Ethernet. The operating system installed on each machine was the Fedora Linux. The urban region in question covered an area of the city of Shanghai, China, with 56,948 streets and 41,689 intersections.

Three experiments were defined in order to measure the scalability, linearity and data flow capacity of the architecture:

- Vehicles demands were set to a fixed value and system performance was tested by varying the amount of participant nodes. For this, a fixed demand with 600,000 vehicles and a synchronization interval of 5 seconds were defined. Tests were performed with the number of nodes ranging between 1 and 6. The computation and communication times are illustrated in Figure 4.3;
- Performance was measured by varying the synchronization interval. In this experiment the number of participant nodes and the demand values (again with 600,000 vehicles) were fixed. Figure 4.4 displays the results. As can be seen, increasing the communication interval – by reducing the amount of times this operation is performed – leads to a performance enhancement, but not substantially. The authors concluded that increasing the interval between synchronizations is not a good strategy to achieve performance gains.
- Performance was again measured by varying the demand values, but fixing the number of nodes and the synchronization interval (set to 5 seconds). The demand ranged between 360,000 and 720,000 vehicles. As can be seen in Figure 4.5, the computation and communication times grow almost linearly in relation to the size of the demand.

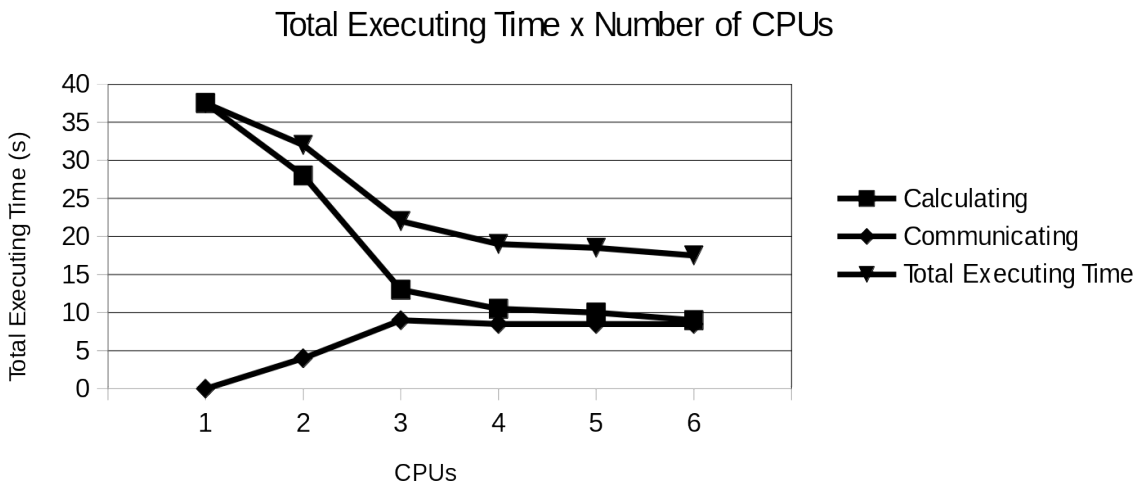


Figure 4.3: *Execution times according to the number of CPUs (adapted from [59]).*

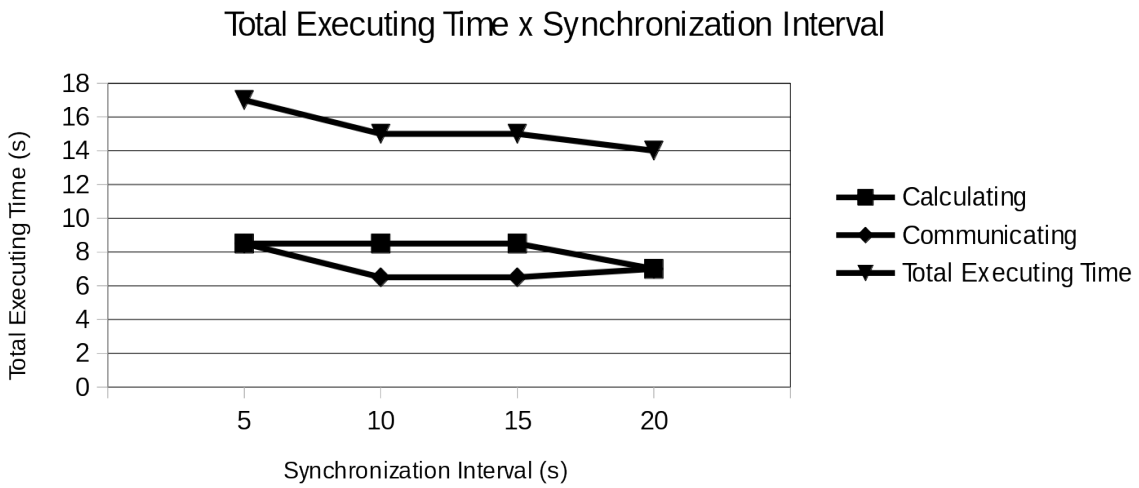


Figure 4.4: *Execution times according to synchronization interval (adapted from [59]).*

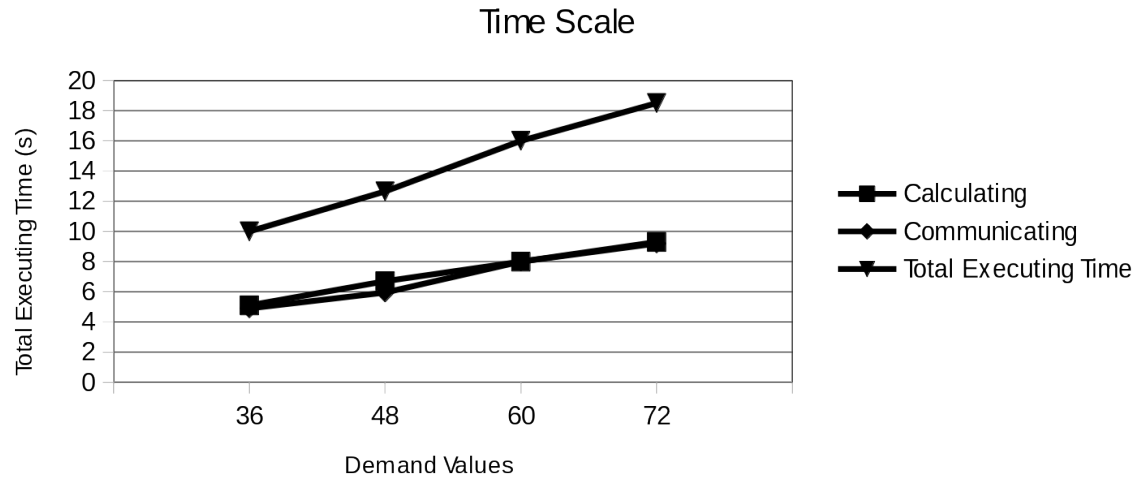


Figure 4.5: *Execution times under diverse demand values (adapted from [59]).*

4.1.2 Dealing with the Network Partition Problem

Dali, Feng and Xinxin [248] focus on one of the central problems of parallel computing: to ensure that the division of the tasks – also called partition or decomposition – leads to highly uniform workloads to the processors involved in the computation. Figure 4.6 illustrates a bad division of the study area, where CPUs 1 and 3 receive little work, while CPU 4 is overloaded.

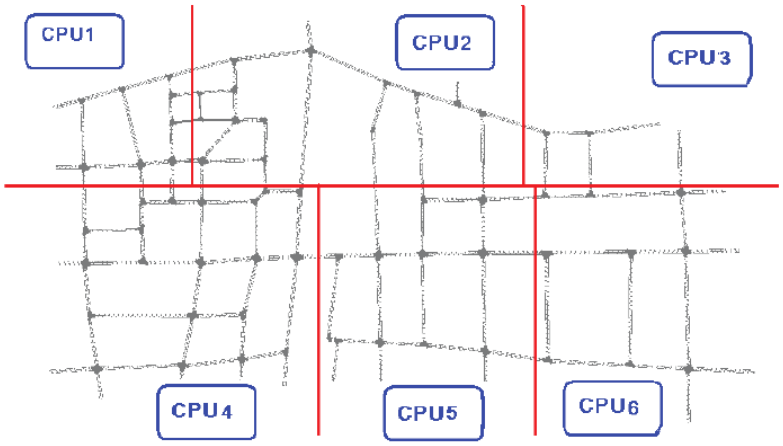


Figure 4.6: *Non-uniform domain decomposition (extracted from [248]).*

An uniform partition of the road network is a fundamental pre-requisite for efficient parallel simulation, since the simulation as a whole depends on the slower processor (the one with the greatest workload). A decomposition is considered effective if it meets two requirements:

1. The workload is well balanced between processors;
2. The time consumed in the communication process is small.

Dali, Feng and Xinxin also argue that another determining factor for good performance of the algorithm is the *workload* (number of vehicles) traveling on a street/avenue during the simulation process. This workload depends not only on length of the avenue or its geographical location but also on the *traffic density* (***d***). The most extensive roads tend to have less workload if ***d*** is small. Traditional partitioning methods – based on the length of the avenue as its initial weight during the decomposition process – lead to an irregular division of workload between computational processors and hence to a lower performance of the simulation.

In their proposal, Dali Feng and Xinxin suggest not to employ the length of street/avenue as a measure of the initial weight, but previously estimate the workload that each road will receive and use that value for the partitioning method. This value is

calculated based on two pre-known data: the origin-destination (OD) demand matrix and the routes chosen by the drivers.

Using such information, the workload of each street/avenue is used as a weight for a recursive bisection method that partitions the road network into sections with workloads roughly equivalent.

To measure the efficiency of the proposed method, two experiments were done:

1. A comparative analysis of the workload when applying the new method and the conventional one. To measure the improvement in load balancing, the efficiency measure $R = \frac{W_{min}}{W_{max}}$ was defined, where W_{min} and W_{max} correspond respectively, to the smallest and largest workload in the sub-regions. The closer to 1, the better the balance;
2. Parallel performance experiments, using the *speedup* and efficiency as indicators.

Test Environment

The evaluation of the proposed method was carried out with a *KD-50-I* high-performance computer as the experimental platform. Among the reasons for choosing this computer, the authors highlighted the fact that it is formed by 336 Loongson-2F CPUs, had low cost, low power consumption, small footprint and the advantage of running only free software, which ensured its constant updating. It is also worth mentioning that the same machine had been successfully used in other traffic studies.

Results

All simulations were performed for half an hour using the same O-D matrix. Table 4.1 shows the comparative results between the traditional and the proposed methods, with several CPU nodes; Figure 4.7 displays the performance curves of the two methods.

Worker Computing Nodes	Load Balance Efficiency	
	Conventional Algorithm	Improved Algorithm
2	0.83	0.97
4	0.66	0.80
6	0.27	0.74
8	0.26	0.64
16	0.12	0.51
24	0.09	0.45
32	0.07	0.36
40	0.04	0.33
48	0.03	0.30
56	0.05	0.16

Table 4.1: *Evolution of the workload according to the number of computing nodes (extracted from [248]).*

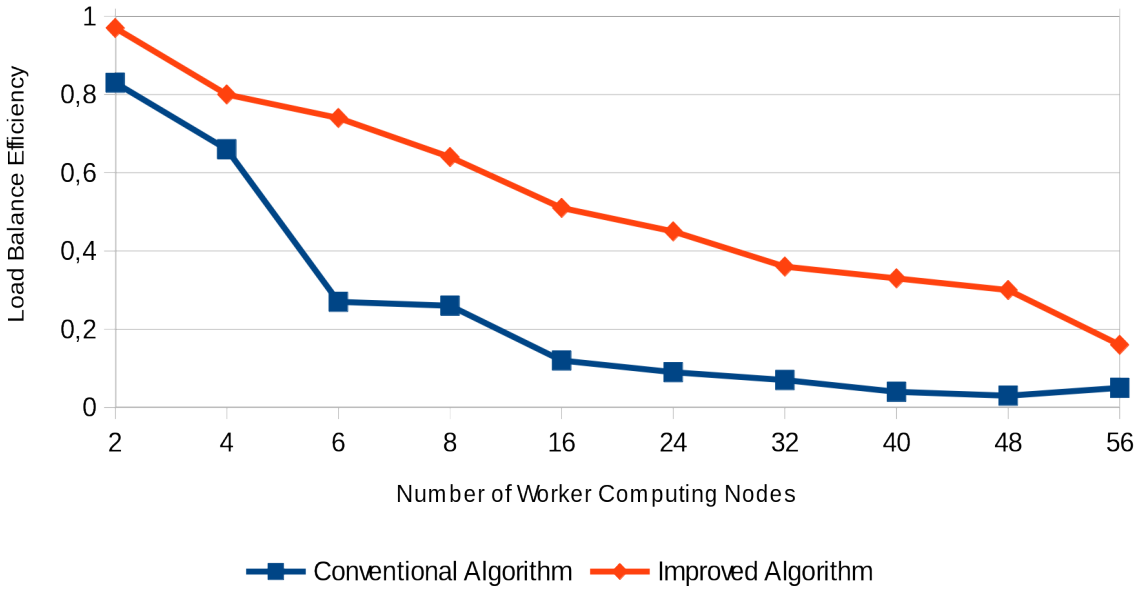


Figure 4.7: *Graphic of the evolution of the workload according to the number of computing nodes (adapted from [248]).*

It can be seen from Figure 4.7 that the efficiency of the proposed algorithm is closer to one than the traditional method, which means that its resulting workload is evenly distributed. Although this value decreases as the number of CPU nodes increases, it happens more slowly and evenly, always keeping ahead of the conventional method.

Figures 4.8 and 4.9 show the workloads of different road sub-regions, when split into 6 CPUs. They show that the greatest workload obtained by the new method is about

200 vehicles smaller than that achieved by the traditional method, thus better distributing the traffic simulation. The algorithm is more centered around 800 vehicles per CPU (test satisfactory average) while the conventional one is more dispersed.

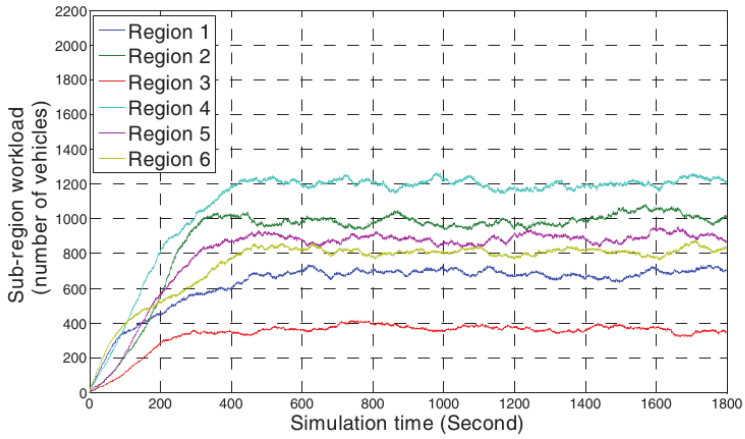


Figure 4.8: *Workload in road sub-regions, varying with simulation time, produced by the conventional algorithm (extracted from [248]).*

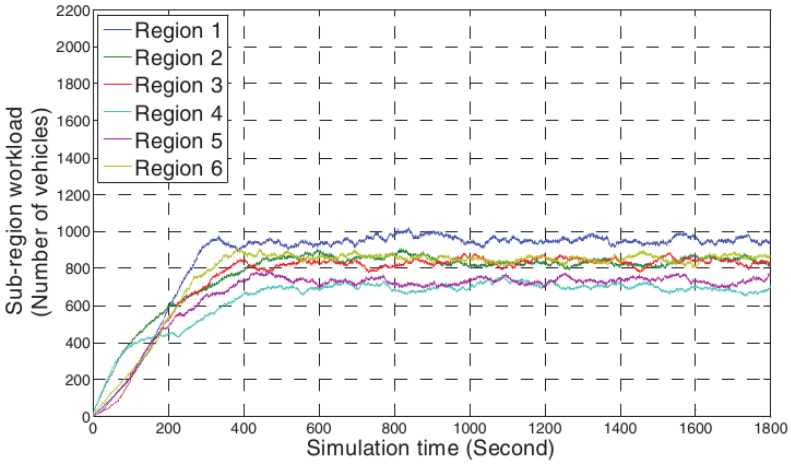


Figure 4.9: *Workload in road sub-regions, varying with simulation time, produced by the new algorithm (extracted from [248]).*

Once shown that the new method provides better load balancing – one of the essential requirements for an efficient parallelization – the authors went on to analyze application performance by using, for this, the following measures:

- ***Speedup*** – defined as $\frac{T_1}{T_p}$,
- ***Efficiency***: Defined as $\frac{T_1}{p*T_p}$;

where T_1 is the time for simulation with only one processor and T_p is the simulation time using p processors. Figures 4.10 and 4.11 display the results in terms of speedup and efficiency, respectively.

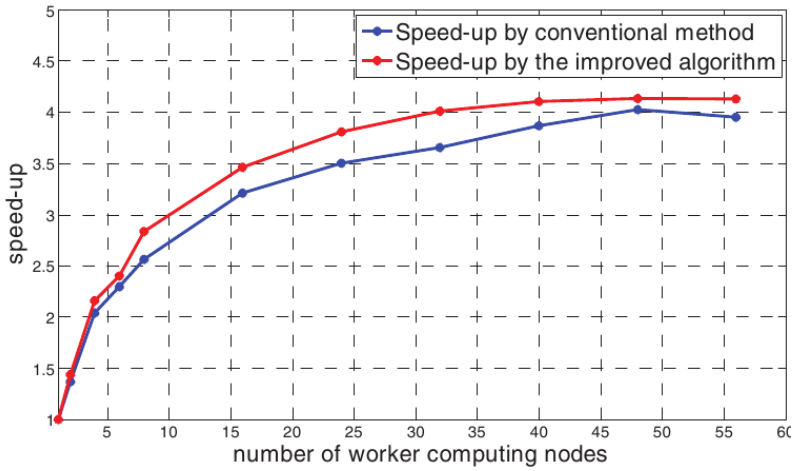


Figure 4.10: *Parallel speedup of two methods (extracted from [248]).*

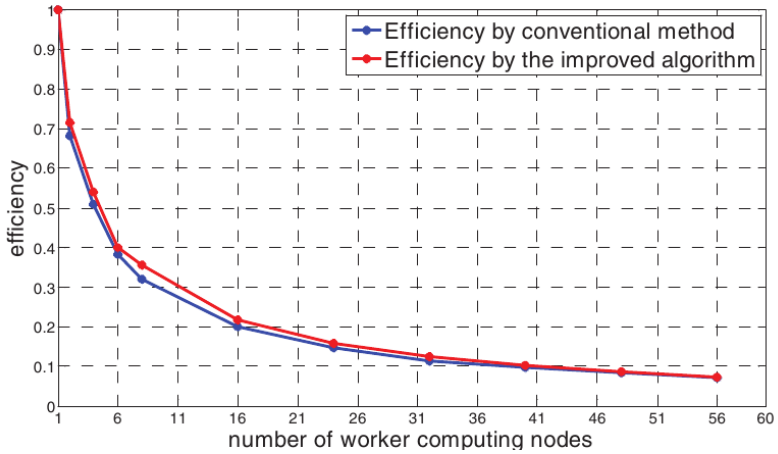


Figure 4.11: *Parallel efficiency of two methods (extracted from [248]).*

It is clear from the figures that the computation times, in both methods, are lower when parallel techniques are used.

Under the speedup and efficiency points of view, the algorithm performance is superior using the new partitioning method, when compared to the traditional one. With 8 CPUs, in the experiments carried out the performance of the parallel simulation became 10.87% higher. Regarding the speedup, the maximum achieved value was 4.136, while the conventional method reached 4.125. This value stabilized around 48 processors and could not be improved because of the rising of communication costs.

4.2 Macroscopic Simulations

Although there is a considerable number of parallel approaches to traffic simulation based on microscopic and mesoscopic models, the same can not be said about macro-

scopic models, which suffer from a serious lack of studies and systems implemented that use these models. Next is presented a brief description of the most significant work in this area that we discovered during the course of this work.

4.2.1 Real Time Macroscopic Simulations

Chronopoulos and Johnston [46] present a parallel mechanism able to generate macroscopic simulations and predict the traffic conditions in real time using a *n*CUBE2 parallel computer. The authors argue that these predictions can be used for real-time traffic control and drivers guidance.

The *n*CUBE2 is a MIMD parallel computer with a *hypercube* topology with distributed memory and communication through message passing. With *ndim* being a positive integer, in a hypercube there are $p = 2^{ndim}$ processors, labeled $0, 1, ..., p-1$. Two processors P_{j1} and P_{j2} are directly connected if the binary representation of j_1 and j_2 differs in exactly one bit. Each edge in a graph of a hypercube is a direct link between two processors. Figure 4.13 displays a graph of a hypercube with *ndim = 4*.

The architecture of the proposed system is shown in Figure 4.12. It consists of the parallel computer system, a data handling system (DHS), a simulation program and two interface devices.

The DHS manages the data structures, the storage of traffic measurement information and physical characteristics of the road network. For this, it can use a database system. The simulation program is the software that implements the macroscopic model in the parallel computer. The interface devices are computing network devices that take care of sending and receiving data for the parallel simulation system.

System Evaluation

Tests with a system implementing the architecture simulated two hours of traffic in a region of the US city of Minneapolis and showed a clear superiority when compared to a sequential algorithm. While the parallel approach was able to complete the simulation in just 5.25 seconds, the sequential one needed 141 seconds for its completion. That is, a speedup of almost 27 times was achieved.

In the tests the authors used, as performance metric, the **runtime**, also called **total execution time (etime)** which is formed by:

1. **Time for data input (itime)** – time needed by the first processor for reading the data from the hard disk and sending them to all other processors;
2. **Computation Time (ctime)** – time spent by the algorithm to perform the traffic simulation;

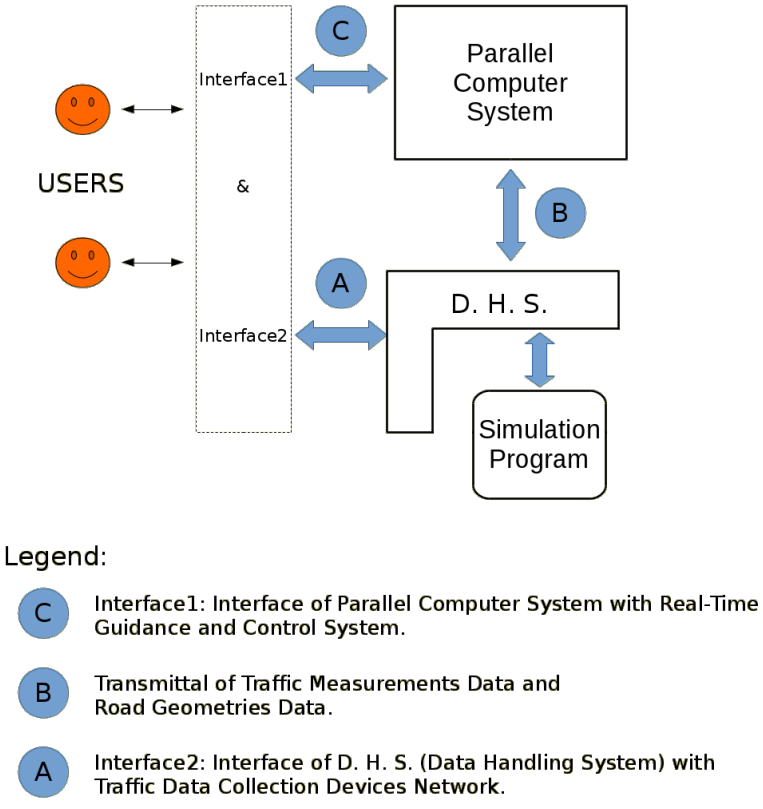


Figure 4.12: *Architecture of the Macroscopic Real Time Simulation System (adapted from [46]).*

3. **Time for data output (otime)** – time to collect the data from all processors by the first processor and to write the results back to the hard drive;

Tables 4.2 and 4.3 display the results of the parallel run times (in seconds) showing that, in settings with a higher number of processors, about 16% of them remain idle. This occurs because the road network is divided into segments (areas) of equal sizes and each segment is then mapped to a processor. Since there are more processors than segments, the phenomenon becomes evident.

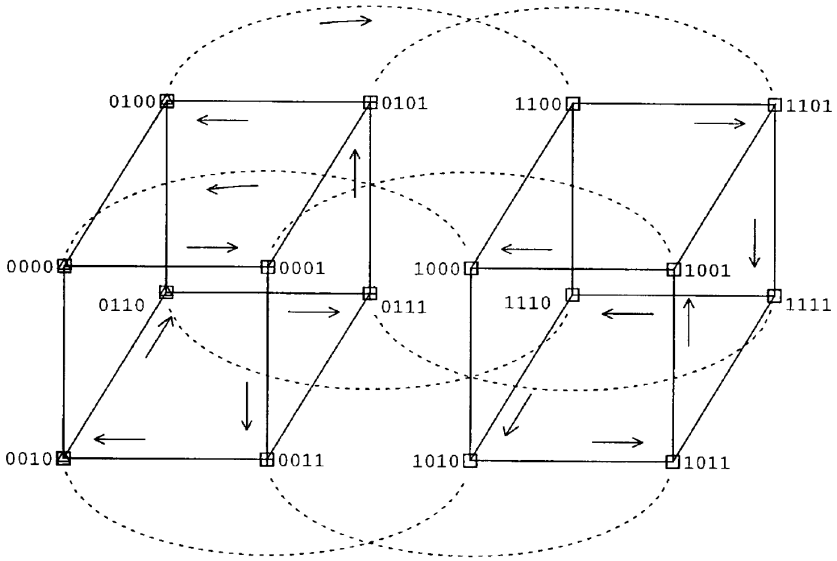


Figure 4.13: *Hypercube with ndim=4 (extracted from [46]).*

Ndim	Used Procs	Max. Procs	etime	itime	ctime	otime
9	425	512	5.250	0.80	4.06	0.44
8	213	256	5.493	0.73	4.39	0.40
7	107	128	6.133	0.73	5.03	0.39
6	61	64	7.166	0.72	6.07	0.38
5	31	32	9.539	0.71	8.40	0.44
4	16	16	13.745	0.71	12.65	0.40
3	8	8	22.438	0.70	21.32	0.41
2	4	4	39.900	0.72	38.66	0.52
1	2	2	72.906	0.70	71.81	0.40
0	1	1	141.012	0.67	139.94	0.40

Table 4.2: *Simulation execution times with 2^9 processors (adapted from [46]).*

Ndim	Used Procs	Max. Procs	etime	itime	ctime	otime
10	851	1024	9.563	0.90	8.12	0.63
9	426	512	10.070	0.80	8.76	0.56
8	213	256	11.312	0.79	10.05	0.50
7	122	128	13.497	0.80	11.97	0.74
6	61	64	17.790	0.76	16.63	0.41
5	32	32	26.319	0.76	25.11	0.46
4	16	16	43.829	0.75	42.61	0.47
3	8	8	77.829	0.76	76.66	0.42
2	4	4	146.508	0.74	145.37	0.40
1	2	2	281.571	0.81	280.33	0.43
0	1	1	554.444	0.71	553.33	0.40

Table 4.3: *Simulation execution times with* 2^{10} *processors (adapted from [46]).*

The experiments proved that the achieved speedup is large enough to justify the use of the developed parallel mechanism as part of a real-time traffic control system.

The maximum speedup was achieved in configurations with more processors. However, once this maximum is reached, the parallel efficiency decreases with the increase in the number of processors. According to the authors, this is inevitable due to the problem size, small when compared to the number of CPUs. If the problem size is expanded, the parallel computing efficiency will probably continue to increase.

4.3 Traffic Simulation on GPUs

Although this is a relatively recent research area, there are already some works related to traffic simulation with the use of GPUs. This section discusses some of them.

Shen, Wang and Zhu

In [226], Shen, Wang and Zhu present a GPU implementation of an agent-based traffic microsimulation, aiming at the optimization of signaled intersections.

In agent-based modeling (ABM), a system is modeled as a set of independent and interacting entities called agents, each one able to take decisions autonomously. Agents individually evaluate their own situation and make decisions based on a collection of pre-established rules [22]. In turn, a multi-agent system (MAS) refers to a computerized simulation formed by multiple interacting agents.

In their implementation, Shen, Wang and Zhu use a GPU parallel genetic algorithm for solving the traffic signal timing optimization problem. After presenting the problem formulation and the overall implementation, the authors test their approach in a road network with four signaled intersections.

The experiments, performed using a PC equipped with one AMD Athlon TM 64 X2 Dual Core processor 4000+ and an NVIDIA GeForce GTX470 GPU, showed a speedup of 195x when compared to its equivalent sequential version.

The authors acknowledge, however, that in order to perform a more realistic evaluation of the proposed strategy, the work must be extended to larger scale road networks.

Sano and Fukuta

In [218], Sano and Fukuta describe a GPU-based multi-agent system framework for large-scale traffic simulations.

The authors point that, in order to improve the reproducibility of real situations, the agents involved in a micro-simulation should respond to dynamic and unpredictable environmental events, such as disasters or sudden climate changes.

To achieve this, agents should be programmed to react to such environmental changes and it is important that, even with this programming, the simulation is still able to run in reasonable time.

Making these considerations, Sano and Fukuta present a GPGPU-based framework that allows to easily perform large scale simulations, accelerating not only the simulation itself, but also the code that has to be implemented in order that agents could respond in real-time to dynamic environmental changes.

After presenting the details of the proposed framework, the authors perform some evaluations of the processing performance to validate its potential scalability, using different GPUs.

For that, an OpenCL-based implementation of the framework was employed. In the experiments, they used a map consisting of 12 nodes and 22 links. The number of agents using the road network ranged from 1 to 2048.

The experiments proved the scalability of the proposed framework, since as the number of agents increased, the time of the equivalent sequential algorithm grew vertiginously faster than its parallel counterpart.

4.4 General Remarks

This chapter has briefly explored how parallelism has been applied to urban traffic simulations. As can be observed, the great majority of approaches focus on micro and mesoscopic simulations, which are naturally very computationally demanding.

In all systems presented, the proposed parallelism usually focuses on the division of a large road network into smaller sub-regions, each one assigned to one processor. There is no mention of the use of parallel approaches to the internal routines of the simulation.

During the performed bibliographic review, there was a great difficulty in identifying scientific works and implemented systems that employed parallelism for macroscopic approaches. Only one system was found, and this dates back to the end of the 90's. Nothing related to macroscopic simulation on GPUs was found, demonstrating the lack of research in this area. This opens space for new proposals that explore parallelism in such context, as the one developed and described in the present work.

The next chapters investigate computational problems that occur in the study of urban traffic conditions and propose GPU-based parallel algorithms to solve them.

CHAPTER 5

A GPU-Based Algorithm for Enumerating All Chordless Cycles in Graphs

Paths and cycles are among the most important and fundamental structures in the study of graphs, and their discovery and/or enumeration becomes essential to solve many computational problems in areas such as optimization (see Chapter 7), concurrent operating systems (for detection of deadlocks) [227, p. 333–337], bioinformatics (an Eulerian path approach is used to reassembly DNA sequences from their fragments) [198], CMOS circuit design (used to search for an optimal gate ordering) [212], information retrieval, natural language processing [20, 71, 81, 82, 137], identification of regions in urban traffic networks that are poorly connected [208] and many others.

The remainder of this chapter is organized as follows. Section 5.1 presents preliminary definitions. Section 5.2 presents the ideas that underpin the sequential algorithm described by Dias et al. [74]. The proposed parallel algorithm is introduced in Section 5.3. Section 5.4 describes the experimental tests and the results produced by the new algorithm. Conclusion and future work are discussed in Section 5.5.

5.1 Background

Consider a finite undirected simple graph $G = (V,E)$, with $n = |V|$ and $m = |E|$. A *chordless cycle* C is an induced subgraph that is a cycle, i.e., apart from the edges of C that form a cycle, E does not contain any other edges that join vertices of C. The graph presented in Figure 5.1, representing a small region of Goiânia downtown network, highlight three of such structures.

Sequential and parallel algorithms for the problem of determining if a graph contains a chordless cycle with $k \geq 4$ vertices, for some fixed cycle of length k, were proposed by Chandrasekharan et al. [41]. They presented a sequential algorithm where a cycle C_l, $l \geq k$, can be found in $O(m^2 \cdot n^{k-4})$ time and a parallel algorithm adopting the CRCW PRAM model (see Appendix A) that demands $O(\log n)$ time using $(m^2 \cdot n^{k-4})$

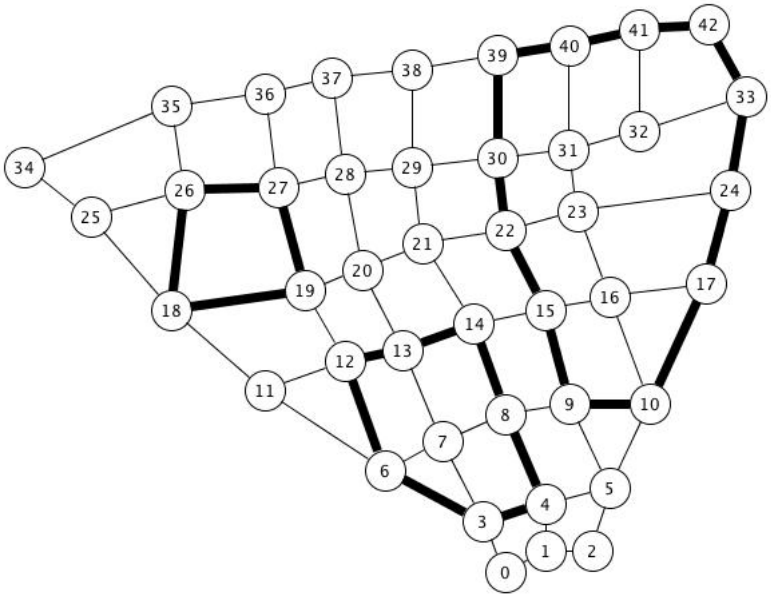


Figure 5.1: *Simple representation of Goiânia downtown network, Goiás, Brasil.*

processors. However, finding just one cycle of length greater than or equal to a fixed value k is easier than enumerating all chordless cycles in a graph G.

In general, the enumeration of particular subgraphs of a given graph belongs to the complexity class $\mathcal{P}$-complete, whose resolution is as hard as the resolution of problems in the $\mathcal{NP}$-complete class [23, 240]. Although there are exact sequential algorithms to solve problems in such a class, they become impractical in cases when the number of structures to enumerate grows exponentially with the size of the graph. This suggests the use of approximation methods (like heuristics and meta-heuristics) or parallel computing. Such approaches do not reduce the complexity of printing the final enumeration, but seek to reduce the construction time of the solution set, either through the relaxation of listing rules (the first option), or the use of multiple execution units (the latter).

A large amount of sequential algorithms have been proposed for enumerating graph structures such as cycles [17, 207, 217, 250], circuits [18, 232], paths [119, 207], trees [146, 207] and cliques [168, 236]. These tasks are usually hard to deal with, since many classes of graphs present the aforementioned problem. Nevertheless, enumeration is necessary in the resolution process of several practical problems. In particular, the enumeration of chordless cycles is useful in some important areas, including:

- *Identification of weakly coupled regions of urban road networks* [145, 208]. Usually, during the occurrence of an unpredictable phenomena (like large-scale car collisions and/or interdictions of urban areas due to floods or earthquakes) there are some network regions in which traffic becomes unfeasible, because traffic can not be re-routed. Chordless cycles represent such regions (called here network holes),

and their identification and study allow traffic managers to define preventive and/or corrective actions in order to deal with such exceptional situations, minimizing their impacts on the transit;

- *Prediction of nuclear magnetic resonance chemical shift values* [220, 239]. In chemistry, graphs representing chemical structures usually have a large number of cycles (called rings) and their distribution and size patterns are essential information that directly affect important physical properties and their chemical and biological reactivity;
- *Study of ecological networks with the aim of identifying predators that compete for the same prey* [73, 229]. Here, a directed *food web* graph is transformed into a *niche overlap* graph to represent the competition between species. The lack of chordless cycles in this latter graph means that the species can be rearranged along a single hierarchy. In the example depicted in Figure 5.2, since Great Kiskadee and Nightingale prey the same animal (small frogs), in the niche-overlap graph they are connected by an edge.

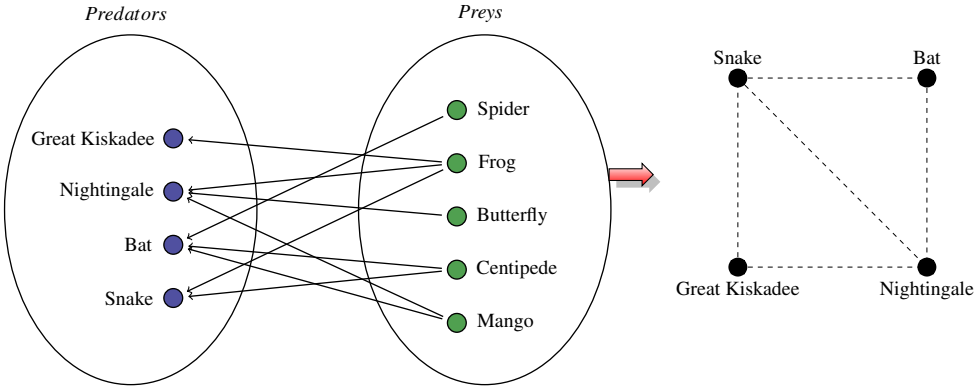


Figure 5.2: *Transforming a food web graph into a niche-overlap graph.*

To the best of our knowledge, the fastest sequential algorithm for enumerating all chordless cycles in any undirected simple graph developed until 2012 was the one proposed by Sokhn et al. [229]. The general principle of this algorithm is to use vertex ordering and to expand paths from each vertex using a depth-first search (DFS) strategy. This approach has the disadvantage of finding each chordless cycle twice. Unfortunately, the authors did not present its complexity analysis.

In 2013 Dias et al. presented a new sequential algorithm for enumerating all chordless cycles [74]. That algorithm finds each chordless cycle just once in $O(n+m)$ time for each chordless cycle and is significantly faster than Sokhn et al.'s method.

In 2014, Uno and Satoh [239] presented another sequential algorithm for the same problem. However, their method also repeats chordless cycles in the output. Actu-

ally, each chordless cycle appears as many times as its length, leading to the time complexity of $O(n \cdot (n+m))$ for finding each one of them.

In another study, Ferreira et al. [86] presented algorithms for listing all $\boldsymbol{C}$ chordless cycles and *st*-paths in undirected graphs in $\tilde{O}(m+n \cdot C)$ time.

Although the algorithm developed by Dias et al. [74] is able to enumerate all chordless cycles without repetition and, in terms of execution speed, surpasses all other chordless cycle enumeration algorithms known to us, it still takes a considerable processing time when applied to some classes of complex graphs and to graphs whose chordless cycles grow exponentially in graph size.

Again, as far as we know, a previous practical parallel algorithm for the problem of enumerating all chordless cycles in an undirected graph does not exist. In this chapter, we make a first step towards filling this gap in the literature by presenting a GPU-based parallel chordless cycle enumeration algorithm that is fast when applied to difficult graphs.

5.2 Mathematical Definitions

We now present some mathematical definitions that support our approach to enumerate all chordless cycles of a graph. For further details, see Dias et al [74].

Let $G = (V, E)$ be a finite undirected simple graph with vertex set V and edge set E. Let $n = |V|$, $m = |E|$, $Adj(x) = \{y \in V \mid (x,y) \in E\}$ be the set of neighbors of a vertex $x \in V$ and $Adj[x] = \{x\} \cup Adj(x)$ be the closed neighborhood of x.

A *simple path* is a finite sequence of vertices $\langle v_1, v_2, \ldots, v_k \rangle$ such that $(v_i, v_{i+1}) \in E$ and no vertex appears repeated in the sequence, that is, $v_i \neq v_j$, for $i, j \in \{1, \ldots, k-1\}$ and $i \neq j$. A *cycle* is a simple path $\langle v_1, v_2, \ldots, v_k \rangle$ such that $(v_k, v_1) \in E$. We denote a cycle with k vertices by C_k. A *chord* of a path (cycle) is an edge between two vertices of the path (cycle), that is not part of it. A path (cycle) without chords is called a *chordless path (chordless cycle)*.

The minimum and maximum degrees, among all vertices of G, are denoted by $\delta(G)$ and $\Delta(G)$ (or simply δ and Δ), respectively. The degree of a particular vertex $v \in V$ is denoted by $d_G(v)$. The subgraph induced by the subset $V - X$, for $X \subseteq V$ ($V - \{u\}$, for $u \in V$), is denoted by $G - X$ ($G - u$). The degree of a particular vertex $v \in V$ is denoted by $d_G(v)$.

An ordering of the vertices of G can be defined by a bijection $\ell : V \to \{1, 2, \ldots, n\}$. We call this bijection a *vertex labeling*.

Note that, if G has a cycle C with k vertices, then it can be represented in $2 \cdot k$ ways, in clockwise or counterclockwise and by all its possible rotations, expressed as

$\langle v_{i-1}, v_i, v_{i+1}, \ldots, v_k, v_1, v_2, \ldots, v_{i-2} \rangle$ and $\langle v_{i+1}, v_i, v_{i-1}, \ldots, v_2, v_1, v_k, \ldots, v_{i+2} \rangle$. However, if we impose the following constraints:

1. the labeling of the second vertex of the cycle has to be smaller than the labeling of all other vertices ($\ell(v_2) = \min\{\ell(v_i) \mid i = 1, \ldots, k\}$);
2. the labeling of the first vertex of the cycle has to be smaller than the labeling of third vertex ($\ell(v_1) < \ell(v_3)$).

then any cycle can be defined in a unique way. The proof for this statement is simple. Let v_i be the vertex of the cycle with the smallest labeling. The representation of the cycle can be rotated until v_i becomes the second vertex in the sequence. Now there are two possible representations for that cycle, clockwise and counterclockwise. Since the neighbors of v_i in the cycle are v_{i-1} and v_{i+1}, exactly one of these possibilities satisfies condition 2.

In the approach introduced by Dias et al. [74], a vertex labeling is characterized by a particular bijection $\ell : V(G) \rightarrow \{1, \ldots, n\}$ called *degree labeling*. It is constructed over a sequence of subgraphs of G, starting with $G_1 = G$. For $i \geq 1$, the $(i+1)^{\text{th}}$ subgraph is defined as $G_{i+1} = G_i - u_i$, for a chosen $u_i \in V(G_i)$ such that $d_{G_i}(u_i) = \delta(G_i)$. Given this sequence, the degree labeling is defined as $\ell(u_i) = i$ for each i.

A *triplet* is defined as a sequence of three vertices that can initiate a chordless path of length greater than three, already following the two aforementioned constraints. Let $T(G)$ denote the set of all initial valid triplets of G, that is, $T(G) = \{\langle x, u, y \rangle \mid x, u, y \in V$ with $x, y \in Adj(u)$, $\ell(u) < \ell(x) < \ell(y)$ and $(x, y) \notin E\}$. The above labeling scheme and the way of formally defining the triplets enable the algorithm proposed by Dias et al. [74] to find every chordless cycle only once and to begin with a smaller initial set of chordless paths, which significantly reduces the search space. Because of the degree labeling, if G is a tree then there are no possible triplets ($T(G) = \varnothing$). For other types of graphs many triplets may exist, even for distinct paths of the same cycle. As detailed in [74], an upper bound for the initial search space size is given by $|T(G)| \leq \frac{(\Delta-1) \cdot m}{2}$.

Given a chordless path $p = \langle v_1, v_2, \ldots, v_k \rangle$ and a vertex $v \in Adj(v_k)$, $v \neq v_{k-1}$, exactly one of the following occurs:

1. $\langle p, v \rangle = \langle v_1, v_2, \ldots, v_k, v \rangle$ is a chordless path;
2. there exists $i \in \{1, \ldots, k-1\}$ such that $p = \langle v_i, v_{i+1}, \ldots, v_k, v \rangle$ is a chordless cycle.

Since $v \in Adj(v_k)$, $v \neq v_{k-1}$ and p is a chordless path, then $\langle p, v \rangle$ is a simple path that extends p. Suppose that $\langle p, v \rangle$ is not a chordless path, that is, there is $i \in \{1, \ldots, k-1\}$ such that $(v, v_i) \in E$. Choosing i^* the biggest index i with this property, we have the mentioned chordless cycle. Case 1 states that path $\langle p, v \rangle$ is an expandable chordless path.

Case 2, with $i^* \neq 1$, states that path $\langle p, v \rangle$ has a chord[1] or, with $i^* = 1$, then $\langle p, v \rangle$ is a desired chordless cycle.

5.2.1 The Sequential Approach

The sequential algorithm for the enumeration of chordless cycles of Dias et al., whose pseudo-code is presented in Algorithm 5.1, is briefly described here in order to promote the understanding of the proposed parallel approach. Further detail and experimental results can be found in [74].

A degree labeling is initially calculated for the input graph G (Line 1). Then, the set $T(G)$ of initial valid triplets (Line 2) is computed. The set C of cycles is initialized (Line 3) with all triangles (which are also chordless) and the set $T(G)$ is assigned to a set T of expandable paths (Line 4).

Next, starting with the initial triplets, a DFS strategy is used for incrementally creating and expanding the set of chordless paths, until each one becomes a chordless cycle or is simply discarded. An expanded path $\langle p, v \rangle$ is dropped when the addition of v to p results in a chord or when the restriction $\ell(v) > \ell(v_2)$ is violated.

[1] Obviously, $\langle v_{i^*}, v_{i^*+1}, \ldots, v_k, v \rangle$ for i^* is also a chordless cycle. But it will be discarded at the enumeration process, since it appears in the expansion of another path p.

Algorithm 5.1: *SequentialChordlessCycles(G)*

DegreeLabeling
Input: Graph G.
Output: Set C of all chordless cycles of G.

1 **perform** *DegreeLabeling(G)*;
2 $T(G) \leftarrow \{\langle x,u,y \rangle \mid x,u,y \in V : x,y \in Adj(u); \ell(u) < \ell(x) < \ell(y)$ and $(x,y) \notin E\}$;
3 $C \leftarrow \{\langle x,u,y \rangle \mid x,u,y \in V : x,y \in Adj(u); \ell(u) < \ell(x) < \ell(y)$ and $(x,y) \in E\}$;
4 $T \leftarrow T(G)$;
5 **while** $(T \neq \varnothing)$ **do**
6 $\quad p \leftarrow \langle v_1, v_2, \ldots, v_k \rangle \in T$;
7 $\quad T \leftarrow T - \{p\}$;
8 $\quad$ **foreach** $v \in Adj(v_k)$ **do**
9 $\quad\quad$ **if** $((\ell(v) > \ell(v_2))$ **and** $(v \notin Adj(v_i), i \in \{2, \ldots, k-1\}))$ **then**
10 $\quad\quad\quad$ **if** $v \in Adj(v_1)$ **then**
11 $\quad\quad\quad\quad C \leftarrow C \cup \{\langle p, v \rangle\}$;
12 $\quad\quad\quad$ **else**
13 $\quad\quad\quad\quad T \leftarrow T \cup \{\langle p, v \rangle\}$;
14 **return** C.

In Lines 9 and 10, the expansion of a path $\langle v_1, v_2, \ldots, v_k \rangle$ by the addition of a neighbor v of v_k is verified and may result in one of three cases:

1. a chordless cycle; or
2. another expandable path; or
3. a chord in the current path or a path that does not respect the labeling constraints.

In case 1, the newly found chordless cycle is added to the set C (Line 11); in case 2, the expanded path is added to the set T (Line 13); in case 3, the path is discarded. The same process is repeated until the set T becomes empty.

Due to the initial conditions of the triplets and the way in which the search is performed, the algorithm finds all chordless cycles and yet avoids rotations of the same solution (two or more cycles with the same structure but that start at different vertices). This provides a faster execution.

Dias et al. [74] presented another version of their method that uses a specialized breadth-first search (BFS). It has some properties that ease the complexity analysis of the algorithm, but adds an overhead to the total computation time.

A possible strategy for the parallelization of Algorithm 5.1 is the expansion of multiple chordless paths through the simultaneous checking of the feasibility of augmenting every path in T with each neighbor of its last vertex. This is adopted in the current thesis and the details are described next.

5.3 The Proposed GPU Algorithm

In this section, we present our parallel algorithm for the problem of chordless cycle enumeration. The adopted approach was to split Algorithm 5.1 into two stages and define a parallel strategy for each.

The first stage involves the construction of sets C and $T(G)$ (Lines 2–4 of Algorithm 5.1). The second stage takes each path $\langle v_1, v_2, \ldots, v_k \rangle$ in $T(G)$, that characterizes a chordless path, and tries to expand it by adding a neighbor v to the last vertex (Lines 5–13).

The computation of the degree labeling (Line 1 of Algorithm 5.1) was not parallelized. Due to its inherent sequential nature and to the low impact in the processing time of the algorithm, this step was kept sequential as a preprocessing task and the resulting labels were used in the later parallel stages.

The proposed parallel algorithm was mapped to a GPU architecture, which follows the basic concepts presented in Chapter 3. In the next section, problems with the data structures of the sequential algorithm are discussed and new data structures for the parallel approach are presented. After that, details of the two parallel stages are described.

5.3.1 Data Structures

Usually, graphs are represented by adjacency matrices or adjacency lists. Although an adjacency matrix enables the verification of connectivity between two vertices in constant time, it has three primary issues:

- for sparse graphs, it wastes a significant amount of memory space;
- due to the large space occupied, it is not possible to allocate the entire matrix in the fast, but small, GPU SM's local memory. Even in advanced models, this memory does not exceed 64KB. A simple graph containing just 256 vertices would be enough to fill up this memory ($256 \cdot 256 \cdot 1$ byte $= 65536$ bytes) with such a data structure;
- exclusively using the GPU's global memory leads to poor performance of the algorithm, because its access time is much higher than that of the SM's local memory (see Section 3.2 for a general overview on modern GPUs).

Consequently, the use of adjacency lists, which allows a more compact graph representation, is justifiable. However, the variable size of each vertex list still does not provide an efficient implementation with GPUs.

To overcome such problems, we used an adapted version[2], with three vectors V_e, E_e and L_v, of the compact graph representation proposed by Harish and Narayanan [123]. Vector V_e is associated with the vertices of a graph $G = (V,E)$. A V_e index is the original vertex identification and the corresponding vector content indicates the position of its first neighbor in the adjacency vector E_e. Since the graph is undirected, it is necessary to represent each edge $(i, j) \in E$ in the adjacency lists of both i and j. So, $|E_e| = 2 \cdot |E|$. Vector L_v stores the degree labels associated with each vertex of G.

If the lists of adjacent vertices are kept sorted in E_e, a binary search can be used to check, in time $O(\log \Delta)$, whether two vertices are adjacent.

Based in graph presented in the Figure 5.1, this compact representation is illustrated in Figure 5.3. Using 2 bytes for an adjacency index, the representation takes only $(|V| + |E|) \cdot 2 \cdot 2$ bytes. This is small enough to store graphs of several sizes, specially if they are sparse, as the urban traffic networks, in the fast local memory of each SM in the majority of GPUs currently available. The search time for listing the neighbors of a vertex in this data structure is $O(\Delta)$, even for dense graphs.

To allow the efficient storage of partial and complete solutions (chordless paths and chordless cycles, respectively), a map of bits was employed. A single bit is enough to indicate whether a vertex belongs to a solution because it is not important to store the vertices in the order that they occur in the chordless paths or cycles. This map is defined

[2]The original version contains only the vectors V_e and E_e. Our adapted version includes the vector L_v.

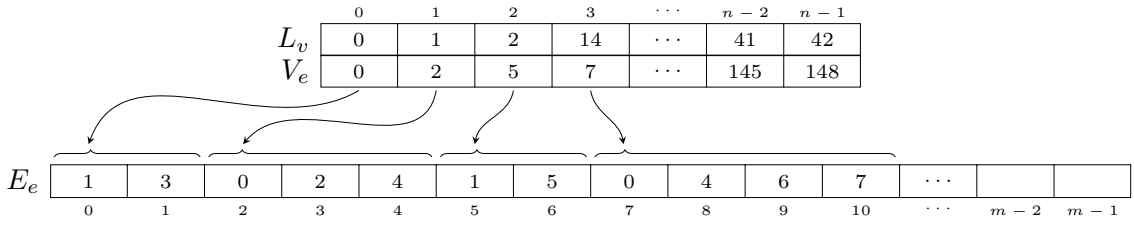


Figure 5.3: *Compact representation of a graph.*

by a bi-dimensional matrix S that contains a row for each chordless path or cycle and n columns of bits, one for each vertex of the graph. The number of bytes necessary for encoding each column is $\lceil \frac{n}{8} \rceil$. Vertex v_j belongs to path or cycle i if, and only if, bit j of row i is 1. Despite the fact that such bitmap does not provide the vertex order in any cycle, it depicts unambiguously each chordless path or cycle in G. Besides the small required space, this data structure enables the addition of a vertex to a solution by a simple bitwise operation, what is computationally inexpensive.

In Fig. 5.4, row 0 contains a combination of bits that represent a chordless cycle in a graph G with $n \leq 24$. In this case a path or cycle storage requires only 3 bytes.

However, with this matrix, it is not possible to identify neither the latest vertex added to a chordless path, nor its initial or second vertex. These vertices are essential to the algorithm. The last vertex is used for expanding the path, while the initial vertex facilitates a check whether the path forms a chordless cycle, and the second vertex is part of a labeling condition check.

To circumvent this problem, three auxiliary vectors, V_1, V_2 and V_L, are used. Vectors V_1 and V_2 store the first and the second vertex of the paths, respectively, and the contents of their cells never change once they are set. V_L stores the last vertex added to the chordless paths. Its content is updated whenever a path is expanded. The sizes of S, V_1, V_2 and V_L have to be sufficiently large to contain information about all chordless

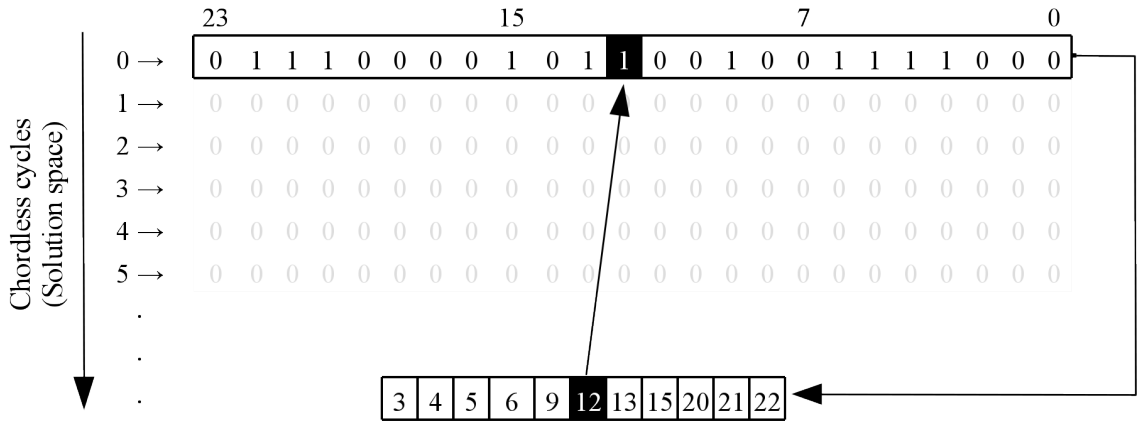


Figure 5.4: *Solution Space, where each vertex occupies just one bit.*

paths that are being processed at any moment.

When the number of rows in each vector or matrix equals the number of chordless paths, these data structures can potentially require a large memory space. Thus, they are kept in the global memory of the GPU. Further, as we describe in Section 5.3.3, these data structures are replicated in order to speed up the processing of chordless paths.

5.3.2 First Stage

The first stage involves the parallelization of Lines 2 to 4 of Algorithm 5.1, which compute the sets C and $T(G)$. Our parallel approach for this stage is condenssed in Algorithm 5.2. It consists of starting $M = (|V| \cdot \Delta^2(G))$ parallel threads in the GPU. Each thread j uses its unique global identifier $gId(j)$ to compute the indices of a triplet $\langle x, u, y \rangle$ in the compact graph representation (Lines 2 to 4 of Algorithm 5.2):

$$i_u \leftarrow \left\lfloor \frac{gId(j)}{\Delta^2} \right\rfloor, \tag{5-1}$$

$$i_x \leftarrow \left\lfloor \frac{gId(j) - i_u \cdot \Delta^2}{\Delta} \right\rfloor, \tag{5-2}$$

$$i_y \leftarrow gId(j) \bmod \Delta, \tag{5-3}$$

where i_u is the index of vertex u in the vector V_e; i_x and i_y are relative indices of x and y in the vector E_e. Index i_u ranges from 0 to $|V| - 1$, and i_x and i_y range from 0 to $\Delta - 1$.

Values i_x and i_y are used to determine two neighbors of the vertex u. They have to be added to the value $V_e[i_u]$ in order to obtain absolute indices in E_e. However, i_x and i_y should be employed only if they refer to valid neighbors (that is, if they are less than or equal to the number of adjacent vertices of u). Such analysis is carried out in Algorithm 5.2 in Lines 5 to 10. The functions $neighborsLowerBound(u)$ and $neighborsUpperBound(u)$ return, respectively, the absolute indices of the first and of the last neighbors of u in E_e, enabling validation of the indices i_x and i_y (Lines 8–9). These two lines use an expression evaluation strategy for avoiding a conditional (if) command when setting the values of x and y. This is faster and most appropriate to the GPU architecture than regular conditional statements.

Finally, with valid vertices u, x and y, each thread tests the label condition $\ell(u) < \ell(x) < \ell(y)$ and continue to be executed only if this label condition is satisfied. The algorithm also checks whether or not x is a neighbor of y and, if so, the triplet $\langle x, u, y \rangle$ is added to the set C. Otherwise, the triplet is added to the set $T(G)$.

Lines 2 to 12 of Algorithm 5.2 require constant time, while Line 13 is $O(\Delta)$. Lines 14 and 16 require serialization in the index calculation of the last used position in

Algorithm 5.2: *FindInitialTripletsParallel*(G)

Input: Compact representation of an undirected simple graph $G = (V,E)$.
Output: Sets C and $T(G)$.

1 **for** *each thread* j, $j = 0,\ldots,|V|\cdot\Delta^2 - 1$ **do in parallel**
2 $\quad i_u \leftarrow \left\lfloor \frac{j}{\Delta^2} \right\rfloor$;
3 $\quad i_x \leftarrow \left\lfloor \frac{j - i_u\cdot\Delta^2}{\Delta} \right\rfloor$;
4 $\quad i_y \leftarrow j \bmod \Delta$;
5 $\quad k_1 \leftarrow neighborsLowerBound(u)$;
6 $\quad k_2 \leftarrow neighborsUpperBound(u)$;
7 $\quad u \leftarrow i_u$;
8 $\quad x \leftarrow (-1)\cdot(i_x > (k_2 - k_1)) + (E_e[k_1 + i_x])\cdot(i_x \leq (k_2 - k_1))$;
9 $\quad y \leftarrow (-1)\cdot(i_y > (k_2 - k_1)) + (E_e[k_1 + i_y])\cdot(i_y \leq (k_2 - k_1))$;
10 $\quad$ **if** $((x \neq -1)$ ***and*** $(y \neq -1))$ **then**
11 $\quad\quad \ell(x) \leftarrow L_v(x)$; $\ell(u) \leftarrow L_v(u)$; $\ell(y) \leftarrow L_v(y)$;
12 $\quad\quad$ **if** $((\ell(u) < \ell(x))$ ***and*** $(\ell(x) < \ell(y)))$ **then**
13 $\quad\quad\quad$ **if** $x \in Adj(y)$ **then**
14 $\quad\quad\quad\quad C \leftarrow C \cup \{\langle x,u,y\rangle\}$;
15 $\quad\quad\quad$ **else**
16 $\quad\quad\quad\quad T(G) \leftarrow T(G) \cup \{\langle x,u,y\rangle\}$;

order to write $\langle x,u,y\rangle$ into C or $T(G)$ at the right position. In the worst case, $O(|V|\cdot\Delta^2)$ threads may try to perform such writing operations simultaneously, but the experiments carried out show that this rarely occurs, even with large graphs. Moreover, this serialization is much faster than other computations performed by the algorithm since it only reserves a free memory position to write a chordless path or cycle. The writing operation, by itself, is done in parallel.

The total time complexity of Algorithm 5.2 is $O(\Delta) + O(|V| + \Delta^2)$, where the second term is due to the serialization and has a low hidden constant.

5.3.3 Second Stage

The second stage of our approach, described in Algorithm 5.3, parallelizes Lines 5 to 13 of Algorithm 5.1. It uses all the processors of the GPU in parallel for evaluating the possibility of expanding the chordless paths computed in Stage 1 (and saved in $T(G)$).

This is done by allocating a thread for every processor and making each thread consider the feasibility of expanding a chordless path $p = \langle v_1,\ldots,v_t\rangle$ with one of the possible Δ neighbors of its latest vertex (v_t). If the number of processing elements

$(|SM| \cdot MaxSMSize)$[3] is greater than or equal to $|T| \cdot \Delta$, then all possible expansions for every path p are analyzed in parallel in one single execution of lines 5 – 15. Otherwise, some threads will repeat this work for the other non-analyzed combinations of paths and Δ neighbors, as controlled by the "*while*" loop in Line 4.

Lines 5 and 6 of Algorithm 5.3 define which chordless path p will be processed by thread j. Lines 7 to 10 specify the neighbor v of v_k. If v_k has less than Δ neighbors, then there will be some exceeding threads. Such threads satisfy the condition $v = -1$, in Line 11, and nothing needs to be done. Finally, Lines 12 to 15 perform a task according to two cases that are similar to what we have in Stage 1:

Algorithm 5.3: *ExpandChordlessPathsParallel*(G, ℓ)

Input: Compact representation of an undirected simple graph $G = (V, E)$ and list ℓ of labels.

Output: Sets T and C of chordless paths and cycles.

```
1  gSize ← |SM| · MaxSMSize;
2  for each thread j, j = 0,...,gSize − 1, do in parallel
3      Pos ← j;
4      while (Pos < |T| · Δ) do
5          i_p ← ⌊Pos/Δ⌋;
6          p ← getCurrentPath(T, i_p);
7          k_1 ← neighborsLowerBound(v_t);
8          k_2 ← neighborsUpperBound(v_t);
9          i_v ← Pos mod Δ;
10         v ← −1 · (i_v > (k_2 − k_1)) + (E[k_1 + i_v]) · (i_v ≤ (k_2 − k_1));
11         if (v ≠ −1) and (v ∉ p) and (L_v(v) > L_v(v_2)) then
12             if (v ∈ Adj(v_1)) and (v ∉ Adj(v_i), i ∈ {2,...,k − 1}) then
13                 C ← C ∪ {⟨p, v⟩};
14             if (v ∉ Adj(v_i), i ∈ {1,...,k − 1}) then
15                 T′ ← T′ ∪ {⟨p, v⟩};
16         Pos ← Pos + gSize;
```

1. If v is adjacent to v_1 but not to other vertices in $\langle v_2, \ldots, v_{k-1} \rangle$, $\langle p, v \rangle$ is a cycle and is added to C;
2. If v is adjacent only to v_k, $\langle p, u \rangle$ is a new expanded path and is saved in a new solution map T'.

[3] As pointed out in Section 3.2, *MaxSMSize* is the maximum number of work-items every SM can handle simultaneously.

Some implementation details of our algorithm are now explained. Firstly, every extended path $\langle p,v\rangle$ is added to T' instead of to T. We do that because it is faster to build a new data structure (for holding the extended chordless paths) than having to update T. In the latter case, it would be necessary to remove $\langle p\rangle$ from T in addition to adding $\langle p,u\rangle$ to this set. Secondly, we use the concept of *persistent threads* [117] to perform the work when there are more combinations of $|T|$ paths versus Δ neighbors than parallel processors. As explained before, the loop at Line 4 does this job, by iterating the analysis for a new combination of path and neighbor vertex.

When the processing of all threads terminates, they have to be restarted for working on the new set T. This task is carried out by a host process, running on the CPU, that replaces T by the recently created T', and launches all threads again. Note, however, that we do not implement the stop condition in the host as a check $T' \neq \varnothing$. This would lead to constant communication between CPU and GPU, significantly degrading the performance of the algorithm. Instead, it is preferable to use a simpler approach, which has shown to be faster: to restart all threads $|V|-3$ times. This number of steps is sufficient, since every chordless path is increased with a new vertex of V, moved to the set C or simply discarded, at each iteration of the host loop. Besides, no path or cycle can have more than $|V|$ vertices. Algorithm 5.4 illustrates the host process. It performs Stage 1 and Stage 2 of our approach.

In Algorithm 5.3, the loop at line 4 iterates at most $\lceil \frac{|T|\cdot\Delta}{|SM|\cdot MaxSMSize} \rceil$ times. Line 5 and Lines 7 to 11 require constant time. Line 6 copies a chordless path from the GPU global memory to a private thread memory. Since $\lceil \frac{|V|}{8} \rceil$ bytes are necessary to store the path, this line takes time $O(|V|)$. Lines 12 and 14 have time complexity $O(k \cdot log(\Delta))$ for a given chordless path p and neighbor v under analysis, because it has to perform $O(k)$ adjacency checks ($k \leq |V|$), each one of them requiring $O(\log\Delta)$ verifications. Lines 13 and 15 are $O(1)$ in theory, but they depend implicitly on synchronized written operations on C and T, similarly to what happens with Lines 14 and 16 of Algorithm 5.2. In the worst case, $|SM| \cdot MaxSMSize$ threads would try to access one of these sets at the same time.

Therefore, the total worst-case time complexity of Algorithm 5.3, as a single thread execution of Line 4 at Algorithm 5.4, is $\lceil \frac{|T|\cdot\Delta}{|SM|\cdot MaxSMSize} \rceil \cdot (O(k\cdot\log\Delta) + O(|SM|\cdot MaxSMSize)) = O\left(\frac{|T|\cdot\Delta\cdot k\cdot\log\Delta}{|SM|\cdot MaxSMSize}\right) + O(|T|\cdot\Delta)$.

We note that term $\boldsymbol{k}$ is the index of the latest vertex of a path $\boldsymbol{p}$ in T and also gives the size of this path. All parallel work-items work with paths of the same size. Furthermore, the size of the paths increases by one at every iteration $\boldsymbol{i}$ of the for loop of Algorithm 5.4. Actually, $k = i+2$. The second part of the time complexity, $O(|T|\cdot\Delta)$, is due to the serialization process.

Hence, Algorithm 5.4 has time complexity $\sum_{i=1}^{|V-3|} O\left(\frac{|T_i|\cdot\Delta\cdot k_i\cdot\log\Delta}{|SM|\cdot MaxSMSize} + O\left(|T_i|\cdot\Delta\right)\right)$, where $|T_i|$ is the size of the set of chordless paths in iteration i and $k_i = i+2$. Although

Algorithm 5.4: $HostProcess(G, \ell)$

Input: Compact representation of an undirected simple graph $G = (V, E)$ and a list ℓ of labels.
Output: Set C of chordless cycles.

```
1 Create the data structures V, E_e, V_1, V_2, V_L, L, C, T and T'
2 Run Algorithm 5.2;
3 for i = 1, 2, ..., |V| − 3 do
4     Run Algorithm 5.3;
5     Wait all threads to finish;
6     T ← T';
7 Return C
```

such a complexity seems high, the hidden constant for the serialization steps is very low and many threads fall in the case where neither C nor T are updated. Another aspect to note is that $|T_i|$ is not necessarily the same over all iterations of the loop "`for`" in Algorithm 5.4. So, the amount of computation performed can vary in each iteration. This will be illustrated in Section 5.4.

Regarding the space complexity, it is not possible to make a prediction about the amount of space that will be used as the number of cordless cycles is potentially large for certain classes of graphs.

5.4 Computational Experiments

Both parallel and sequential algorithms were coded in the C++ language and compiled using a GNU compiler (g++ version 4.8.2 with parameters "`-O3 -mcmodel=medium -m64 -g -W -Wall`"). The parallel algorithm used OpenCL 1.2 with the AMD Software Development Kit 2.9.1. All experiments were performed on a computer with an AMD FX-9590 Black Edition Octa Core CPU, with clock ranging from 4.7GHz to 5.0GHz, 32GB of RAM, runnning Ubuntu 14.04 64-bits operating system. The computer had a Radeon SAPPHIRE R9 290X Tri-X OC GPU video card, with 4GB of memory. The architecture of such a video card provides 2816 stream processing units and an enhanced engine clock of up to 1040Mhz. Its memory is clocked at 1300MHz (5.2GHz effectively).

In order to evaluate the benefits of the parallel algorithm over the sequential one, in terms of processing time to enumerate all chordless cycles, we performed experiments with 23 graphs, divided into three groups.

The first group consists of ten graphs presented in well known databases of ecological studies [49]. These graphs, which have already been considered by Sokhn

et al. [229] represent *food webs*. For the application of such graphs in the current experiments, it was necessary to transform them into undirected *niche overlap* graphs. This was done using the definitions provided by Wilson and Watkins [251].

The second group consists of modified graphs of the urban traffic network of the cities of Sioux Falls (North Dakota, USA [158]), Kochi (Japan [205]) and part of the downtown area of the city of Goiânia, the capital of the state of Goiás, in Brazil. All streets and roads were modeled as undirected edges for the aim of finding chordless cycles.

Finally, the last group contains graphs representing a cycle, a wheel, bipartite graphs and some grid graphs.

Table 5.1 presents details of each graph. It shows the name of the graph, the numbers n and m of vertices and edges, respectively, and the maximum vertex degree. The remaining columns contain information produced by the algorithms. Column C_3 displays the number of cycles of length three. They are found at the first stage of the sequential and the parallel algorithms. Column $\#clc$ provides the number of chordless cycles with length greater than three. The total number of chordless cycles in each graph is the sum of the values in these two columns.

The sequential and parallel algorithms were run ten times for each graph. The average running times of the ten executions are presented in the table in milliseconds. Column T_{seq} displays the average processing times of the sequential algorithm. The next two columns are the average times related to the parallel GPU algorithm. The first column (T_{par_proc}) contains only the processing time spent by the GPU kernels at the first and second stages, plus the time for the sequential degree labeling preprocessing; the second column (T_{par_total}) has the total time of the parallel code; this includes the processing time (T_{par_proc}) plus the communication time between the host and the GPU in order to transfer the graph structure and the solution set C. The last column of Table 5.1 is the speedup of the parallel algorithm over the sequential algorithm (given by $\frac{T_{seq}}{T_{par\_total}}$).

5.4.1 Analysis of the results

The benefits of the parallel algorithm over the sequential one depend on the nature of the graph. As we can see, the speedup is, in general, proportional to the number of chordless cycles, with speedups ranging from $12\times$ to $153\times$ for the most complex cases (with $|C| \geq 100.000$). When the graph does not have many chordless cycles, the sequential algorithm runs faster than the parallel GPU code.

An example where parallelism can not be efficiently applied is shown in Figure 5.5. In this graph, after the labeling step, the only initial triplet is given by $1 \rightarrow 0 \rightarrow 5$. At each step of the expansion stage there will be only one neighbor of last vertex to be

Name	n	m	Δ	C_3	#clc	T_{seq}	T_{par_proc}	T_{par_total}	*Speedup*
CrystalD	24	86	14	293	0	0.333	0.182	0.622	0.536
ChesUpper	37	85	15	167	0	0.370	0.160	0.656	0.564
Narragan	35	168	22	586	0	0.548	0.197	0.709	0.773
Chesapeake	39	90	11	157	0	0.150	0.188	0.700	0.214
Michigan	39	175	27	587	0	0.614	0.197	0.698	0.879
Mondego	46	206	24	886	0	0.725	0.207	0.773	0.938
Cypwet	71	842	46	8946	0	6.417	0.258	0.892	7.196
Everglades	69	1214	56	15627	710	12.407	0.388	1.478	8.395
Mangrovedry	97	2132	80	30659	27426	102.475	1.822	6.510	15.741
Floridabay	128	3249	98	62389	85976	366.495	2.518	15.095	24.279
Goiânia	43	75	4	5	9311	39.594	0.216	3.081	12.849
SiouxFalls	24	76	5	2	176	1.339	1.138	1.812	0.739
Kochi	140	200	7	16	1820137	291811.6	1230.032	9366.160	31.16
C_{100}	100	100	2	0	1	0.149	0.165	0.770	0.193
Wheel 100	101	200	100	100	1	0.225	0.778	1.229	0.183
$K_{8,8}$	16	64	8	0	784	0.473	0.197	0.599	0.790
$K_{50,50}$	100	2500	50	0	1500625	600.661	4.867	10.391	57.805
Grid 4×10	40	66	4	0	1823	15.430	0.185	1.993	7.742
Grid 5×6	30	49	4	0	749	2.610	0.167	1.249	2.090
Grid 5×10	50	85	4	0	52620	199.132	1.982	12.718	15.658
Grid 6×6	36	60	4	0	3436	7.889	0.203	1.570	5.025
Grid 6×10	60	104	4	0	800139	2906.009	6.284	18.989	153.034
Grid 7×10	70	123	4	0	8136453	36955.470	54.840	286.212	129.119
Grid 8×10[a]	80	142	4	0	71535910	427091.02	4655.147	8697.081	49.107

[a] Due to high memory consumption for storing set T when processing Grid 8×10, both the sequential and parallel algorithms were modified to not store the chordless cycles, but only to count them.

Table 5.1: *Average running time to enumerate all chordless cycles on niche overlap graphs and on other well known graphs. Times T_{seq}, T_{par_proc} and T_{par_total} are presented in milliseconds.*

analyzed and, therefore, only a single chordless path will be expanded. Thus, just one work-item will do useful work, while all others will remain idle.

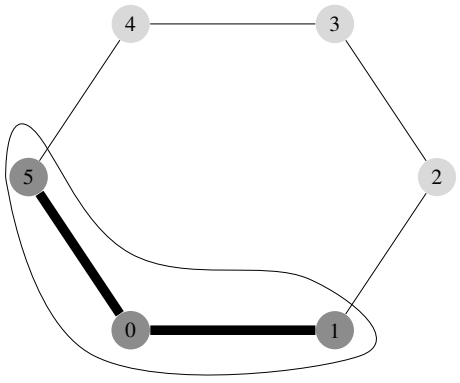


Figure 5.5: *C6: A graph where paralelism is not feasible*

Note, however, that in almost all worst cases (when the speedup was less than 1, indicating a better performance by the sequential algorithm), the most expensive activity

in the parallel algorithm was the data communication between the host and the GPU device, given by $T_{par\_total} - T_{par\_proc}$. So, when considering only the GPU kernel time (column T_{par_proc}), the parallel algorithm is very competitive. Moreover, the parallel algorithm executed in less than 0.002 seconds for all non-competitive cases.

It is useful to see, as well, the evolution of sets C and T in size during the execution of the two stages of the parallel algorithm. This gives a hint about the amount of computation performed by the parallel threads over time, and how much synchronization was necessary for writing on the data structures that hold such sets. Figure 5.6 shows this evolution for the graphs Floridabay, Mangrovedry, Grid 7×10 and Goiânia. The blue (darker) line in each chart represents the size of set T at each call of the kernels; the red (lighter) line shows the change on the size of set C. The X axis represents the results of both stages and also implies the size of all paths in the current set T. Step 1 in the chart refers to the result of the first stage of the GPU algorithm. The next steps, on the right, are related to the output of each iteration (kernel call) of the second stage.

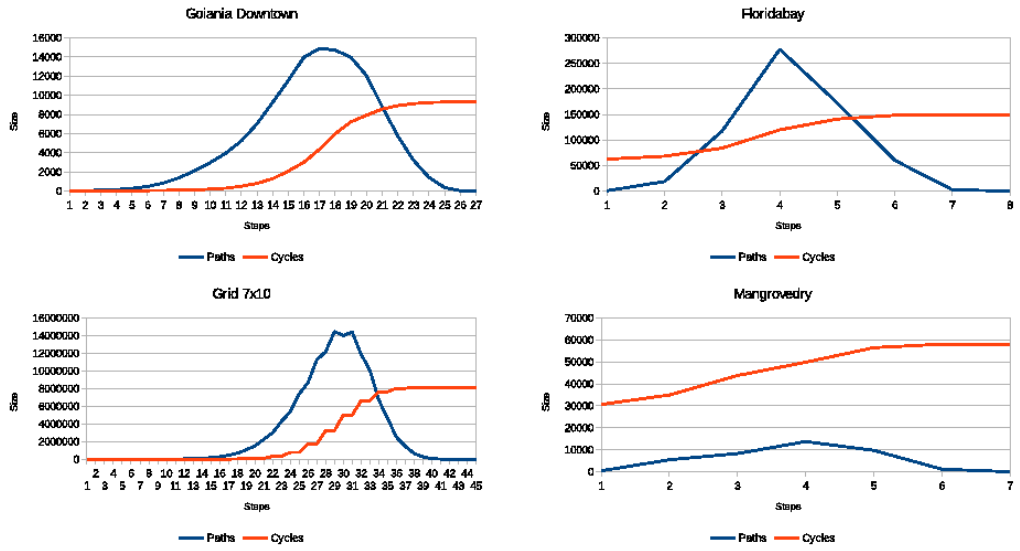


Figure 5.6: *Sizes of T and C for four graphs.*

Both C and T sets are initialized by the first stage of the parallel algorithm. As the algorithm procedes through the second stage, new chordless paths are created by extending smaller paths with adjacent vertices and the set T size increases. In this case, more synchronization for writing in T and C occurs. Later, the expansion of some paths result in chordless cycles (that are then added to C) or in cycles with chords (that are discarded). The overall process ends up giving a wave shape to the evolution chart of T and a less steeply increasing curve to the evolution chart of C.

A curious case was the graph Mangrovedry. Many chordless cycles of size three (around 30.000) were found at the first stage of the parallel algorithm. The second stage of the algorithm performed only seven steps (similarly to graph Floridabay), which resulted

in cycles with at most 9 vertices (recall that all initial chordless paths have length 3 and grow at most one vertex at each iteration of the second stage). Interestingly, the size of T did not change rapidly and stayed always below $|C|$, but C doubled in size.

Regarding processing time, even with a very high peak in the size of T, as far as graph Grid 7×10 is concerned, with 14 million chordless paths, the performance of the parallel algorithm was much superior to that of the sequential one (with a speedup of $\approx 129\times$ for that case).

5.5 General Remarks

This chapter presented a parallel algorithm for GPUs to enumerate all chordless cycles of a given simple, undirected graph. The algorithm is based on a previous work done in collaboration with the author of this thesis, which resulted in an already fast sequential algorithm for the same problem. The parallel algorithm works in two stages and takes advantage of the GPU architecture. A compact data structure for graph representation, distinct types of memories and the persistent thread technique were employed to allow more efficient usage of the GPU memory and the processing units.

Experiments were carried out with several graphs and they showed that the benefits of the parallel algorithm depend on there being a large number of the chordless cycles and chordless paths in the input graph. For graphs with more than 100000 chordless cycles or paths, the speedup of the parallel algorithm over the sequential one was between ≈ 12 and 153. The cases for which the parallel algorithm was worse (took longer than the sequential algorithm) were the ones with very few chordless cycles. For those base cases, our implementation still took less than 0.002 seconds to find all the chordless cycles and most of the exceeding time was spent in data transfer between the CPU and the GPU.

As far as we know, this is the first parallel GPU-based algorithm for the problem of enumerating all chordless cycles reported in the open literature. Note, however, that memory size on a GPU is still a limiting factor since the data structures cannot be larger than the maximum supported structure size. Such hardware constraints limit the size of the problems and solutions that can be dealt with by the GPUs. Thus, as future work, one could develop a new data transportation protocol between the ordinary RAM memory and the GPU memory in order to open space when necessary and allow the enumeration of chordless cycles for much larger graphs. Another future work would be to devise a parallel algorithm for computing the degree labeling. Deleting a vertex during such a computation can lead to a major change in the graph (the decrease of one unit of the degree of every adjacent vertex), indicating that the labeling process has an inherent sequential nature. However, it is possible to update the degree of all vertices in parallel in constant time using $n \cdot \Delta$ processors. Then, the smallest degree can be found through a parallel reduction

in time $O(\log(n))$ with n threads. Repeating this process $n-1$ times provides the desirable result with total time $O(n \cdot \log(n))$.

CHAPTER 6

A Fast and Generic GPU-Based Parallel Reduction Implementation

Reduction operations are extensively employed in many computational problems. A reduction consists of, given a finite set of numeric elements, combining into a single value all elements in that set, using for this a *combiner function* (also known as *associative operator*) like addition, multiplication or finding the largest/smallest element, among others. A parallel reduction, in turn, is the reduction operation concurrently performed when multiple execution units are available.

Widely used as a basic subroutine for a number of algorithms such as Counting Sort [51], Stream Compaction [16], Golden Section and Fibonacci Methods [149] and Radix Sort [51, Chapter 8.3], parallel reduction is also extensively employed in the present thesis. It appears in two steps of the macroscopic assignment algorithm described in Chapter 8, including a shortest path method presented in Chapter 7.

The current chapter, hence, reports an investigation on this subject and depicts a GPU-based parallel approach for it. Employing techniques like *Loop Unrolling*, *Persistent Threads* and *Algebraic Expressions* to avoid thread divergence, the presented approach was able to achieve a 2.8x speedup when compared to [39] and performance equivalent to the best strategy proposed by [124], using a generic, simple and easily portable code.

Experiments conducted to evaluate the approach show that the strategy is able to perform efficiently in AMD and NVidia's hardware, as well as in OpenCL and CUDA.

The remainder of this chapter is structured as follows. Section 6.1 presents the basic definitions of the problem. Section 6.2 briefly describes the techniques currently in use. Section 6.3 explains our approach. Section 6.4 details the experimental environment and the results. Finally, Section 6.5 gives general remarks about the presented strategy.

6.1 Background

Formally, a reduction can be defined as follows [195]: Given a set X with n values, $X = \{x_0, x_1, ..., x_{n-1}\}$, compute $x_0 \otimes x_1 \otimes ... \otimes x_{n-1}$. The associative operator $\otimes$

can be (but is not limited to) any one of the set $\{+, \times, \wedge, \vee, \oplus, \cap, \cup, \max, \min\}$.

Algorithm 6.1: *Summation(A)*

Input: A set $A = \{a_1, a_2, \ldots, a_n\}$ of numeric elements
Output: The sum of all elements
$accumulator \leftarrow 0$
for $i \leftarrow 1$ ***to*** n **do**
  $accumulator \leftarrow accumulator + a_i$
return *accumulator*

Consider the pseudo code shown in Algorithm 6.1. At first glance, it seems that the algorithm is inherently sequential, since the variable *accumulator* depends on the value computed in the previous step, preventing any attempt of parallelization. However, it is possible to avoid this problem by making use of two basic properties of addition and multiplication operations: *Associativity* and *Commutativity*[1].

- **Associativity** means that, given three or more numbers, they can be linked in any order without changing the final result. Taking the sum as an example, it's possible to do $a_1 + a_2$ and, then, add a_3, and the result will be the same as doing $a_3 + a_2$ and then adding a_1. Formally, we have $(a_1 + a_2) + a_3 \equiv a_1 + (a_2 + a_3)$;
- **Commutativity** ensures that no matter the order in which an operation on two numbers a_1 and a_2 is performed, the result will always be the same. Formally, for multiplication, we have $a_1 \cdot a_2 \equiv a_2 \cdot a_1$.

Considering that the order in which the elements are combined does not affect the final result[2, 3], these two properties can be used, dividing the problem into smaller subproblems and these, in turn, solved in parallel. After solving each subproblem, the

[1] Other two properties, *Neutral Element* and *Closeness*, guarantee, respectively, that any number added to zero results in the number itself, and when we add/multiply two or more numbers within the same set (natural, for example), the result will always be a number within the same set.

[2] Although, mathematically, this is true for numbers in any set, in computational terms things are a little more complicated. For instance, these properties hold for the set of integers, but the same does not happen for the floating point numbers due to the inherent imprecision that arises when combining (adding, multiplying, etc.) numbers with different exponents, which leads to the absorption of the lower bits during the combine operation. As an example, mathematically the result of $(1.5 + 4^{50} - 4^{50})$ is always the same, no matter the order the terms are added, whereas the floating point computed value can result in 0 or 1.5, depending on the sequence in which operations are performed [68, 109, 129, 178].

[3] Note that, although this is a complicating factor when a large numerical precision is necessary, it did not actually preclude its application in the golden ratio method (see Section 8.1.2) in GPU because, in the performed experiments, the accumulated error using single-precision floats did not exceed 10^{-5} when compared to its equivalent sequential version running on the CPU using double precision floats. On the other hand, if such precision becomes necessary, the problem could be greatly minimized by adopting the use of double-precision floating points (which potentially can decrease the application performance for certain GPU models) or using some strategies to reduce truncation errors, like the one proposed by Kahan [143], among others.

partial results are combined to produce the final result. Figure 6.1 illustrates the process using the associative operator "+" in an array with 16 elements.

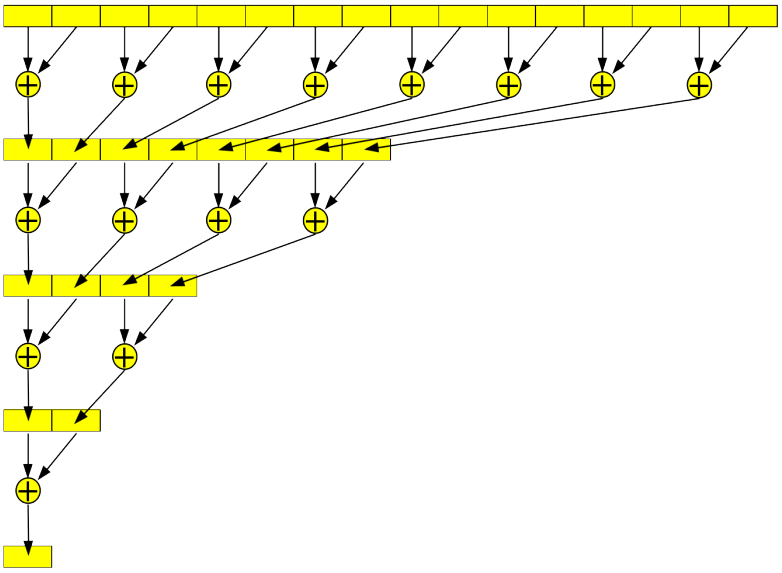


Figure 6.1: *Parallel reduction – associative reduction tree.*

6.2 Parallel Reduction in GPUs

Since the arrival of programmable GPUs, some strategies to accelerate the reduction operation on such devices have been proposed. The two most well known are those described by Mark Harris [124] and Bryan Catanzaro [39]. Most recently, Justin Luitjens [167] presented some improvements to the strategies described in [124]. Unfortunately, the strategies adopted by [124] and [167], although very efficient, are limited to hardware and software provided by NVidia, restricting their use.

On the other hand, the proposal of Catanzaro [39] is based on the open standard OpenCL [116], adopted by a myriad of manufacturers, what makes it portable. Nevertheless, the code presented in [39] also has a weakeness, as it does not adopt some strategies that could significantly improve its performance.

This section details how the *associative* and *commutative* properties can be used to implement efficient parallel reductions on GPUs. As highlighted at the end of Section 6.1, the basic idea is to "split" the problem into smaller pieces and solve them in parallel. However, the execution environment (GPU hardware) imposes some restrictions that must be considered to maximize the *speedup*. Therefore, the details of how GPUs are organized (see Section 3.2) will dictate the choices from now on.

The approaches of Harris [124] and Catanzaro [39] to deal with reductions in GPUs operate in a pretty similar way, using a tree-based structure.

One of the aspects to be considered is the number of elements in the collection (vector) in which the reduction will be applied. If this amount is sufficiently small and can be stored in the local memory of each SM, then the reduction becomes quite simple. In [39], Catanzaro presents some strategies for this case and conducts performance comparisons between them. Then, after describing how reductions can be efficiently performed in small sets, Catanzaro shifts his focus to the cases in which a large volume of data must be handled. Three strategies are presented and a winner, called "*Two-Stage Parallel Reduction*", is elected. Harris [124] deals only with parallel reduction in large datasets.

Our approach is mainly based on a proposal from Catanzaro [39]. Therefore, a more detailed description of it is presented. First, however, we also give an explanation of the strategies by Harris [124] and Luitjens [167], since some ideas for speeding up the computation came from them. Hence, unlike the rest of the thesis, here their original code is presented, and not just the pseudo code.

6.2.1 Mark Harris' Work

The work presented by Harris [124] focuses on techniques for performing reductions of large data volumes. The author shows, through successive versions of the same algorithm, how bad decisions or an incorrect way of mapping the problem to the target platform can negatively impact the application performance.

Problems like shared memory bank conflict, lack of communication between thread blocks (making it impossible for a kernel to reduce a large array at once) and highly divergent warps are addressed. Starting with a naive version, step by step improvements are described, reaching an implementation 30x faster than the first one. Next, we show how the author achieved such speedups.

Harris performed experiments using a G80 GPU. This video card has a 384-bit memory interface, with a 900 MHz DDR memory, which leads to a theoretic $\frac{384*1800}{8} = 86.4GB/s$ of memory bandwidth[4]. All tests were conducted using a vector with 2^{22} (4M) integer values.

In the first version of the reduction (Kernel 1), whose source code was extracted from [124] and is presented in Listing 6.1, Harris points out some issues: the test in line 11 leads to highly divergent work-items in a wave-front, in addition to the fact that the % operator is very slow. Due to such issues, this version has very poor performance: 8.054ms of execution time and only 2.083GB/s of memory bandwidth being used.

[4]Memory bandwidth basically determines how fast is the memory. Usually, it is measured in gigabytes per second (GB/s). The more bandwidth of the memory and the more it is explored by the running program, the faster the computation.

Listing 6.1: *Parallel reduction – interleaved addressing with divergent branching (kernel 1)*

```
1  __global__ void reduce0(int *g_idata, int *g_odata) {
2   extern __shared__ int sdata[];
3   //Each thread loads one element from global to shared mem
4   unsigned int tid = threadIdx.x;
5   unsigned int i = blockIdx.x*blockDim.x + threadIdx.x;
6   sdata[tid] = g_idata[i];
7   __syncthreads();
8
9   //Do reduction in shared mem
10  for(unsigned int s=1; s < blockDim.x; s *= 2) {
11  if (tid % (2*s) == 0) {
12   sdata[tid] += sdata[tid + s];
13  }
14  __syncthreads();
15 }
16 //Write result for this block to global mem
17 if (tid == 0) g_odata[blockIdx.x] = sdata[0];
18 }
```

Overcoming this problem is quite simple: it is sufficient to replace the divergent branch in inner loop (lines 10 – lines 12 of Listing 6.1) with a strided index which leads to a non-divergent branch (Kernel 2). Listing 6.2 shows the modified code.

Listing 6.2: *Parallel reduction – interleaved addressing with bank conflicts (kernel 2)*

```
1 for (unsigned int s=1; s < blockDim.x; s *= 2) {
2  int index = 2 * s * tid;
3  if (index < blockDim.x) {
4  sdata[index] += sdata[index + s];
5 }
6 __syncthreads();
7 }
```

With this modification the performance of the program improves: now it executes in 3.456ms and uses 4.854GB/s of memory bandwidth.

This solution, however, does not solve another problem: the local (shared) memory bank conflict (see Section 3.2.1) that arises when using this kind of memory access pattern. To solve the indicated issue, Harris replaces the strided indexing in the inner loop (lines 1 – lines 4 of Listing 6.2) with a reversed loop and a work-item-id based index. Listing 6.3 presents the new version of the code (Kernel 3).

Listing 6.3: *Parallel reduction – sequential addressing (kernel 3)*

```
1 for (unsigned int s = blockDim.x/2; s > 0; s >>= 1) {
2  if (tid < s) {
3   sdata[tid] += sdata[tid + s];
4  }
5  __syncthreads();
6 }
```

Kernel 3 excecutes in 1.722ms and uses 9.741GB/s of memory bandwidth.

Note, however, that the code presented in Listing 6.3 still has problems to be solved. Due to command "s = blockDim.x/2" in line 1 and to the "if (tid < s)" in line 2, half of the work-items are idle on the first loop iteration, which is certainly a waste of computational resources. To overcome this problem, Harris suggests to halve the number of wave-fronts (line 5 of Listing 6.1) and to replace the single load (line 6) at the begining of the reduction by two loads. The modified version (Kernel 4) can be seen in Listing 6.4.

Listing 6.4: *Parallel reduction – first add during global load (kernel 4)*

```
1 //Perform first level of reduction,
2 //reading from global memory and writing to shared memory
3 unsigned int tid = threadIdx.x;
4 unsigned int i = blockIdx.x*(blockDim.x*2) + threadIdx.x;
5 sdata[tid] = g_idata[i] + g_idata[i+blockDim.x];
6 __syncthreads();
```

With this modification, the code is excecuted in 0.965ms and it utilizes 17.377GB/s of memory bandwidth.

The next strategy is to use the loop unrolling technique. As pointed out by the author, while the reduction proceeds, the amount of "active" work-items (i.e., the ones doing useful work) decreases and, when $s \leq 32$, only one wavefront remains[5].

According to the way GPUs are internally organized (see Section 3.2), all work-items are SIMD synchronous within a wave-front. This means that, when $s \leq 32$ (or 64, for some GPUs) the work-items don't need to be synchronized and the command "if (tid < s)" in line 2 of Listing 6.3 is no longer necessary because it doesn't save any work. Having made this considerations, Harris unrolls the last 6 iterations of the inner loop, adding a new function called "warpReduce". The improved code (Kernel 5) is presented in Listings 6.5 and 6.6.

[5] 64 work-items in the AMD's hardware.

Listing 6.5: *Parallel reduction – warp reduce*

```
1 __device__ void warpReduce(volatile int* sdata, int tid) {
2  sdata[tid] += sdata[tid + 32];
3  sdata[tid] += sdata[tid + 16];
4  sdata[tid] += sdata[tid + 8];
5  sdata[tid] += sdata[tid + 4];
6  sdata[tid] += sdata[tid + 2];
7  sdata[tid] += sdata[tid + 1];
8 }
```

Listing 6.6: *Parallel reduction – unroll last warp (kernel 5)*

```
1  //inner loop
2  for (unsigned int s=blockDim.x/2; s > 32; s >>=1) {
3   if (tid < s)
4   { sdata[tid] += sdata[tid + s]; }
5   __syncthreads();
6  }
7  if (tid < 32) warpReduce(sdata, tid);
```

This version runs in 0.563ms and the memory bandwidth usage now reaches 31.289GB/s.

In the next step of code improving, Harris recalls that CUDA supports C++ template parameters on device and host functions. This allows the specification of the block size[6] as a function template parameter. Since the changes in code performed in this step are relatively large, they will not be presented here. For details about the adapted code, please see [124]. The improved version (Kernel 6) executes in 0.381ms and uses 43.996GB/s of memory bandwidth.

In the last step of code optimization, sequential and parallel reductions are combined. Here, each work-item loads and sums multiple elements in a tree-based reduction in shared memory. To this, the load and add of two elements (Lines 3 – 6 of Listing 6.4) are replaced by a “while” loop to add as many elements as necessary. The modified version (Kernel 7) of the begining of the reduction is shown in Listing 6.7.

[6] Local size in OpenCL nomenclature.

Listing 6.7: *Parallel reduction – completely unrolled and with multiple elements per thread (kernel 7)*

```
1 unsigned int tid = threadIdx.x;
2 unsigned int i = blockIdx.x*(blockSize*2) + threadIdx.x;
3 unsigned int gridSize = blockSize*2*gridDim.x;
4 sdata[tid] = 0;
5 while (i < n) {
6  sdata[tid] += g_idata[i] + g_idata[i+blockSize];
7  i += gridSize;
8 }
9 __syncthreads();
```

As a result of all these optimizations, the final version of the code runs in 0.268ms and the memory bandwidth usage reaches 62.671GB/s. All these improvements are summarized in Table 6.1.

	Time (ms)	Memory Bandwidth (GB/s)	Step speedup	Cummulative speedup
Kernel 1: interleaved addressing with divergent branching	8.054	2.083		
Kernel 2: interleaved addressing with bank conflicts	3.456	4.854	2.33x	2.33x
Kernel 3: sequential addressing	1.722	9.741	2.01x	4.68x
Kernel 4: first add during global load	0.965	17.377	1.78x	8.34x
Kernel 5: unroll last warp	0.536	31.289	1.8x	15.01x
Kernel 6: completely unrolled	0.381	43.996	1.41x	21.16x
Kernel 7: multiple elements per thread	0.268	62.671	1.42x	30.04x

Table 6.1: *Performance for parallel reduction of 2^{22} integer elements (extracted from [124]).*

6.2.2 Justin Luitjens' Work

In [167] Luitjens shows how a new feature of the NVidia's Kepler (and newer) GPU architecture can be used to make reductions even faster when compared to the strategies presented in [124]: the shuffle (SHFL) instruction.

Usually, work-items inside the same SM use the local (shared) memory when they need to communicate (exchange information). This involves a three-step process: writing the data to local memory, perform a synchronization barrier and then read the data back from local memory. The Kepler and newer architectures implement the *shuffle* instruction, which enables a work-item to directly read private data from another work-

item in the same wave-front. According to the author, there are four main advantages in using this instruction:

- It ultimately allows work-items inside a wave-front to collectively exchange or broadcast data;
- It replaces the three-step process by a single instruction, effectively increasing the bandwidth and decreasing the latency;
- It does not use the local memory at all;
- A sync barrier is implicit in the instruction and, hence, a synchronization step inside a workgroup is not necessary.

Figure 6.2 shows how this instruction can be used to build a reduction tree. As pointed out by Luitjens, this figure only includes the arrows for the work-items actually doing useful work. All work-items are indeed shifting values even though these values are not necessary in the reduction process.

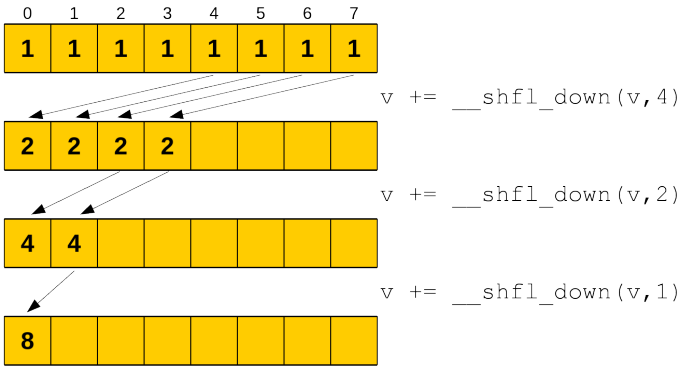


Figure 6.2: *Parallel reduction using the shuffle instruction (extracted from [167]).*

Using this instruction, several versions of the reduction were proposed, implemented and compared. However, although Luitjens states that the adopted strategies lead to faster reductions than those described by Harris [124], no comparative studies between the two approaches were conducted.

6.2.3 Bryan Catanzaro's Work

Now, we describe Catanzaro's two-stage parallel reduction approach for large datasets, as presented in [39].

The technique is based on dividing the data set in p pieces (or "*chunks*"), where p is large enough to keep all GPU cores busy. It is also necessary to limit the number of *work-items* to the maximum amount that the GPU can handle in total without having to switch between them (from now on, that maximum will be called *GS* – or *global size*). Each chunk is then processed by a work-group.

Since the sum operation has the properties of associativity and commutativity, each *work-item* can perform its own reduction sequentially and intercalary with the others. A work-item takes, as the starting point, its global identifier and accumulates, in a private variable, its partial sum, skipping *GS* positions at every step in the vector stored in the GPU's global memory.

After having completed a pass through the data set, the *work-items* in each workgroup write the result of their own reduction in a scrap vector located in local/shared memory which, in turn, will also be reduced in parallel. At the end of the process, each working group will have its own scrap containing, in its position 0, the result of the reduction so far. This partial result is then copied to another vector, this time stored in the GPU global memory, which size must be equal to |SM|. The first stage is then complete. Its source code, extracted from [39], is presented in Listing 6.8.

Listing 6.8: *Two-stage parallel reduction of Catanzaro – stage 1*

```
1  __kernel void reduce(__global float* buffer,
2                       __local float* scratch,
3                       __const int length,
4                       __global float* result) {
5
6    int global_index = get_global_id(0);
7    float accumulator = INFINITY;
8    // Loop sequentially over chunks of input vector
9    while (global_index < length) {
10     float element = buffer[global_index];
11     accumulator = (accumulator < element) ? accumulator : element;
12     global_index += get_global_size(0);
13   }
14   int local_index = get_local_id(0);
15   scratch[local_index] = accumulator;
16   barrier(CLK_LOCAL_MEM_FENCE);
17   // Perform parallel reduction
18   for(int offset=get_local_size(0)/2; offset>0; offset=offset/2) {
19     if (local_index < offset) {
20       float other = scratch[local_index + offset];
21       float mine = scratch[local_index];
22       scratch[local_index] = (mine < other) ? mine : other;
23     }
24     barrier(CLK_LOCAL_MEM_FENCE);
25   }
26   if (local_index == 0) {
27     result[get_group_id(0)] = scratch[0];
28   }
29 }
```

The second stage is simpler. Since now there is a vector with $|SM|$ elements in the global memory – with the result of a partial sum in each position – just the first $|SM|$ *work-items* of the first *SM* copy their corresponding value to an array allocated in local memory. Then the *work-items* perform a new parallel sum of the elements in the vector. After copying the value in position 0 back to global memory, the reduction is finally complete.

Figures 6.3 to 6.7 illustrate these two stages, assuming the presence of two SMs on the GPU, each one able to run four *work-items*.

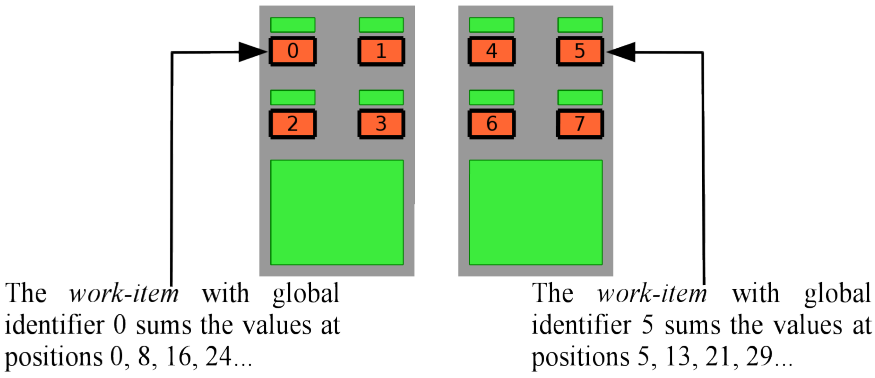


Figure 6.3: *Parallel reduction – first stage, step 1.*

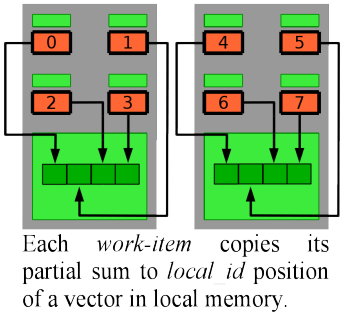


Figure 6.4: *Parallel reduction – first stage, step 2.*

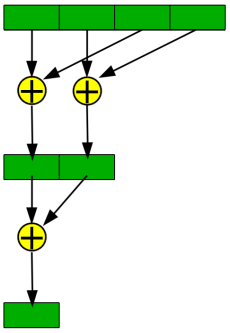


Figure 6.5: *Parallel reduction – first stage, step 3.*

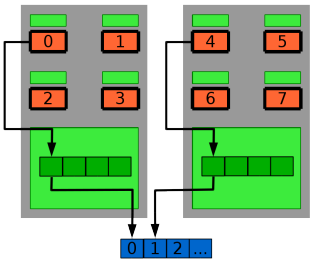


Figure 6.6: *Parallel reduction – first stage, step 4.*

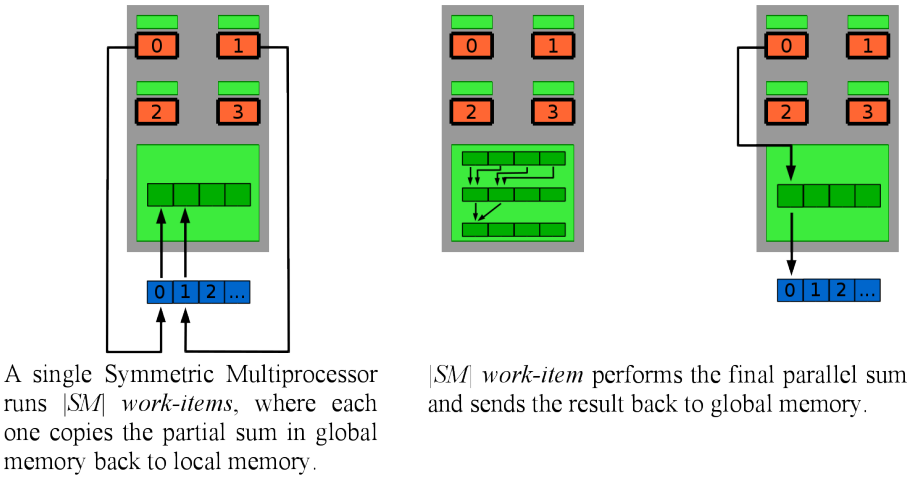


Figure 6.7: *Parallel reduction – second stage, single step.*

6.3 The New Approach

The improvements proposed in our work focus on Steps 1 and 3 of the first stage of the reduction presented in Section 6.2.3. The improvements employ the same strategies proposed by Harris [124] to increase the performance of the approach originally presented by Catanzaro [39] but with appropriately chosen interventions.

In step 1 of the original implementation (Lines 9 – 12 of Listing 6.8), the vector in global memory containing the data to be reduced is entirely traversed by *work-items*, each one performing its own reduction.

This step already uses the "Persistent-Thread" strategy, but its performance can be improved by adopting loop unrolling (see Section 3.3.1). As it can be seen, instead of doing the unroll when the data is in local memory, as proposed by Harris [124] (Listings 6.5 and 6.6 of Section 6.2.1), our improvement performs the unroll in the global memory.

The code presented in Listing 6.9 shows the modified loop, assuming an unrolling factor (F) equal to 4, *iGlobalID* as the *work-item* global identifier and *iLength* as the number of elements to be reduced.

Listing 6.9: *Unrolling the step 1*

```
1  for (iPos = iGlobalID*iUnrollingFactor; iPos < iLength;
2        iPos += iGlobalSize*iUnrollingFactor)
3  {
4   i0 = iPos;    i1 = iPos+1; i2 = iPos+2; i3 = iPos+3;
5   accumulator +=
6   ((i0<iLength)*(aVector[i0])+
7    (i1<iLength)*(aVector[i1])+
8    (i2<iLength)*(aVector[i2])+
9    (i3<iLength)*(aVector[i3]));
10 }
```

A special attention must be given to how the data is brought from the global memory (*aVector*) to the private memory (*accumulator*), through the use of algebraic expressions that prevent reading from invalid memory locations, thus avoiding the usage of "ifs" and potential divergences in the execution flow. The expression $i_n < iLength$ expands to integers 1 or 0 whether it is, respectively, true or false. In the first case $(i_n < iLength) * (aVector[i_n])$ is interpreted as $(1) * (aVector[i_n])$, adding the value stored in location i_n to the partial sum (*accumulator*). In the second case, the expression is interpreted as $(0) * (aVector[0])$, ensuring that – regardless of the data stored in the first position of the vector – value 0 is added to *accumulator*, keeping the partial sum correctness.

At the begining of Step 3, the resulting values of the previous sums are already stored in the local memory of the SMs. Then, each SM performs its own local reduction with its work-items.

In the solutions presented by Harris [124] and Catanzaro [39], in this step all *work-items* are kept synchronized through the use of barriers. However, with minor conceptual changes, it is possible to completely eliminate the overhead caused by the barriers, not only in the last 6 iterations of the loop, as proposed by Harris [124].

Our strategy is to use algebraic expressions to keep all the *work-items* in the same execution step, maintaining its desired behaviour and algorithm correctness.

Consider the highly divergent code presented in Listing 3.4 (Section 3.3.3). Using a simple algebraic expression, it can be rewriten in order to completely eliminate the conditional statement and still return the right result of the comparison, as can be seen in Listing 6.10.

Listing 6.10: *Algebraic "if-then-else"*

```
1 int smallestValue(int a, int b) {
2  return (a < b) * a + (a >= b) * b;
3 }
```

Note that the two boolean operations ($(a < b)$ and $(a >= b)$) are mutually exclusive, being interpreted internally by the compiler as 0 (false) or 1 (true). So, assuming that ***a*** is smaller than ***b***, the result of the algebraic operation is ***(1) * a + (0) * b*** which, ultimately, will return only the value of ***a***.

The same strategy can be applied to lines 18 to 24 of Listing 6.8, that represent the third step of the first stage. The new code is shown in Listing 6.11, where *iLocalSize* stores the number of active local work-items and *iLI* represents the *work-item's* local identifier.

Listing 6.11: *Avoiding Divergences*

```
1 for (iPos = iLocalSize/2; iPos > 0; iPos >>= 1)
2 {
3  bFlag = iLI < iPos;
4  scratch[iLI] += (bFlag)*(scratch[iLI + (bFlag)*iPos]);
5 }
```

Here, in each iteration of the loop, *iPos* is divided by 2 (*iPos > > = 1*) and $bFlag$ is expanded to either 1 or 0, thus reducing by half the number of *work-items* doing a useful job. If, for the current *work-item*, the expression $iLI < iPos$ becomes true, then the expression in the last line will be interpreted as $scratch[iLI] += (1) * (scratch[iLI + (1) * iPos])$, ensuring that the value stored in position $iLI + iPos$ will be added to the value in position *iLI*. On the other hand, if the expression becomes false, it will be interpreted as $scratch[iLI] += (0) * (scratch[iLI + (0) * iPos])$, ensuring that the value in position *iLI* will not be considered. Since all *work-items* are always in the same step of computation – doing exactly the same job (useful or not), independently of being in the same wavefront – sync barriers are unnecessary.

6.4 Computational Experiments

Table 6.2 and Figures 6.8 and 6.9 represent the performance gains achieved against the algorithm described in [39], where $F = 1$ is the runtime of the original code. The machine used in the tests was the same one presented in Section 5.4.

All tests were run on two vectors, one of integers and one of single precision floating points, containing 5533214 elements. There were no measurable differences

between the two vector types.

The times listed in Table 6.2 were obtained with the OpenCL profiler CodeXL, version 2.0.12400.0, and are the averages of five consecutive executions for each $\boldsymbol{F}$.

As can be seen, these results show that the version of the algorithm with $F = 8$ reached a *speedup* pretty close to 2.8x, when compared with the proposal of [39]. It may also be noted that such *speedup* stabilizes around this value ($F = 16$ provided just over 1.5% gain when compared to $F = 8$).

F	Time (ms)	Speedup	Memory Bandwidth (GB/s)	Bandwidth Usage (%)
1	0.249780	1	88.6094002722	26.63
2	0.173930	1.4360949807	127.2515149773	38.24
3	0.139260	1.7936234382	158.9318971708	47.76
4	0.127700	1.955990603	173.3191542678	52.08
5	0.113930	2.1923988414	194.2671464935	58.37
6	0.100810	2.4777303839	219.5502033528	65.97
7	0.093740	2.6646042245	236.1089822914	70.95
8	0.089490	2.7911498491	247.3221142027	74.32
16	0.088160	2.8332577132	251.0532667877	75.44

Table 6.2: *Parallel reduction execution times. New approach compared against Catanzaro's original code.*

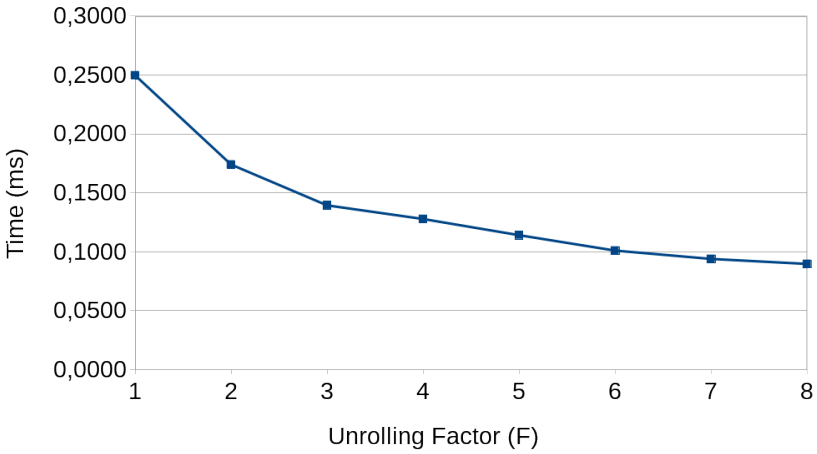


Figure 6.8: *Chart of the parallel reduction execution times.*

The same code was implemented in CUDA and tests were performed against the Kernel 7 of Harris presented in Section 6.2.1. The GPU used in the experiments was a Tesla C2075 with 6GB of memory. The architecture of such a video card provides 448 CUDA cores, a GPU clock of 575MHz and a shader clock of 1150Mhz. Its memory is clocked at 750MHz (3.0GHz effective).

The experiments employed the same two vectors containing 5533214 elements (integers and single precision floating points). Several values of the unrolling factor (F)

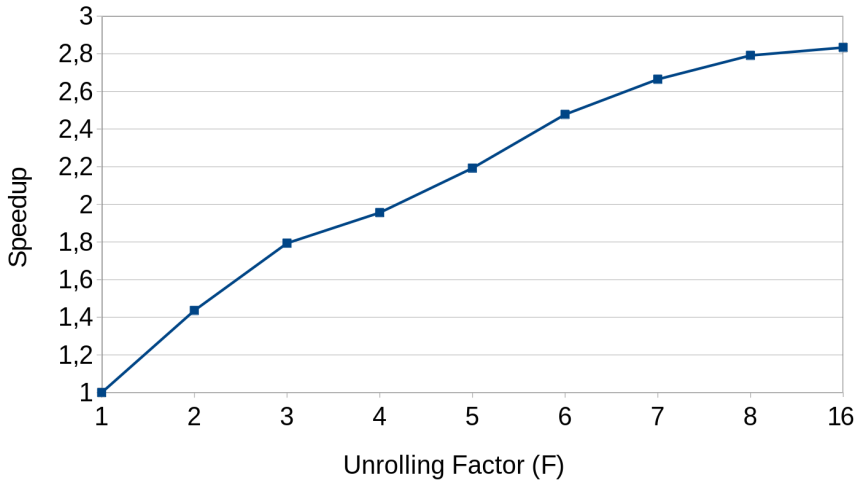


Figure 6.9: *Chart of the parallel reduction speedup.*

were used in order to find the optimal value for such a video board. It was determined that up to $F = 6$ the performance gains were substantial and, with $F \geq 8$, the gains were very discrete. According to this, all experiments were conducted using $F = 8$. Table 6.3 presents the running time (in milliseconds) of both approaches and the percentage of performance (given by the formula $\frac{100 * T_{new}}{T_{k7}}$).

Time – Kernel 7	Time – New Approach	% of Performance
0.17766 ms	0.17867 ms	99.4

Table 6.3: *Parallel reduction execution times – new approach (with unrolling factor equals to 8) compared against Harris' code.*

6.5 General Remarks

Reduction operations are widely employed in many computational problems. This chapter showed how such operations can be performed in a parallel fashion using graphics processing units and detailed the main approaches for them nowadays.

All parallel reduction techniques currently in use suffer from some basic issues. Several only reach their peak performance by employing proprietary strategies and/or technologies, what ends up limiting their use to the platform for which they were designed. Others, though generic, do not adopt certain procedures that could increase their performance without loss of generality.

The strategy presented here combines the best of both worlds: It is generic enough to be used with both CUDA and OpenCL and can run on hardware of the two major GPU manufacturers with minimal changes, just being adapted to the particularities

of each platform. The implemented code, besides simpler, offered a performance equivalent to the best strategy described by Harris [124].

A good performance of this routine is essential for the efficient execution of the macroscopic urban traffic assignment algorithm described in Chapter 8, since it is used on two occasions: in the computation of shortest paths and in the golden ratio method.

CHAPTER 7

A GPU-Based Algorithm for Finding Shortest Paths in Urban Traffic Graphs

As mentioned in Chapter 5, paths are one of the most important and studied structures in graph theory. Its description usually can be found in the first sections of any book on the subject, and a miriad of problems involving paths can be found in literature: routing of telephone and nertwork traffic, navigation through a maze, layout design of printed circuit boards [193], etc. Among them, the problem of efficiently finding the shortest path(s) is one of key importance, either by itself or as a subproblem in more complex tasks.

Thereby, the current chapter presents a study on this problem and proposes a GPU-based parallel approach for it. It is compared with sequential and parallel methods for large graphs. Later, in Chapter 8, the approach discussed here is used as a key step of a more complex algorithm, focused on macroscopic assignment of urban traffic.

The chapter is organized as follows: Section 7.1 presents some basic definitions related to the problem, its four variants, classic algorithms to solve the single source shortest path problem (SSSP) and some parallel algorithms for SSSP. Section 7.2 explains why SSSP is suitable for GPU processing in large urban scenarios and conducts a study on Dijkstra's priority queue behavior. Section 7.3 details the proposed GPU-based Dijkstra algorithm. Section 7.4 describes the experimental tests and the results produced by the new algorithm. General remarks are presented in Section 7.5.

7.1 Background

Extending the definitions in Section 5.2, given a graph $G = (V,E)$ with $n = |V|$ and $m = |E|$, a path in G from a vertex s to a vertex t is a sequence of vertices alternated with edges in the form $p = \langle v_1,(v_1,v_2),v_2,\ldots,(v_{k-1},v_k),v_k\rangle$, where $v_1,v_2,\ldots,v_k \in V$, $(v_i,v_{i+1}) \in E$ for $i = 1,2,\ldots,k-1$, $v_1 = s$ and $v_k = t$. If G does not have multiple edges (two edges in E with the same end points) or if the edges are implied, then p can be written in a more compact way as a sequence of only its vertices.

A path is called *simple* if it does not repeat any vertex. If G is a directed graph then a path p in G is formed by directed edges and p is called an oriented path.

Under a formal definition, a weighted graph G consists of a set V of vertices, a set E of edges and a cost function[1] $w : E \to \mathbb{R}$. Given two vertices s, t in V, the shortest path algorithm finds the path p between s and t with smallest [51]:

$$w(p) = \sum_{i=1}^{k-1} w(v_i, v_{i+1})$$

Figure 7.1 shows a simple graph with the corresponding edge weights and depicts some shortest paths.

Four variants of the shortest path problem (SSP) can be found:

- ***Point to Point (P2P):*** Given $s,t \in V$, respectively source and target vertices of a graph G, the goal is to find a shortest path from s to t;
- ***Single Source:*** Given $s \in V$, compute all shortest paths starting in s to all other vertices of G;
- ***Many to Many:*** For two given sets of vertices $S,T \subset V$ compute the shortest path between all pairs of vertices $(s,t) \in S \times T$;
- ***All Pairs:*** The same as *Many to Many*, but with $S = T = V$.

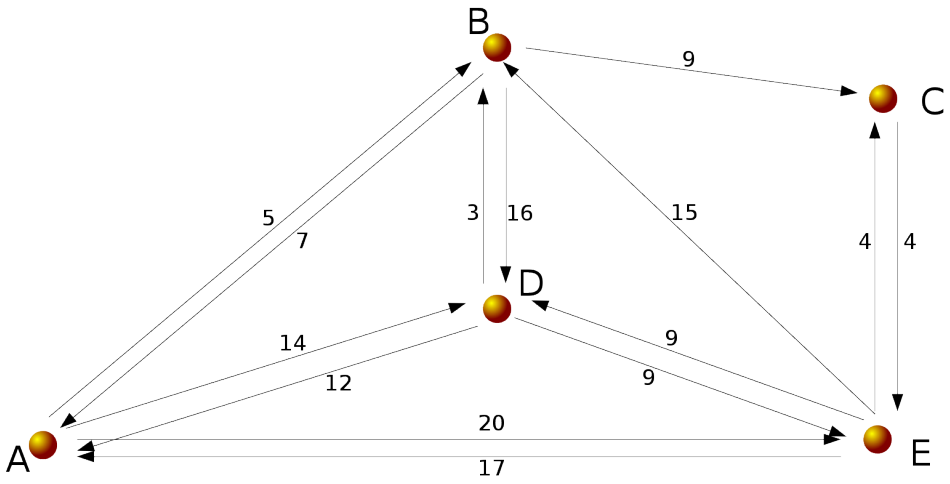


Figure 7.1: *The shortest path between $A \to E$ is $A \to B \to C \to E$, with its cost equals to 18. Between $D \to A$, the shortest path is $D \to B \to A$, costing 10.*

In the next sections the details of the four variants of the SPP, as well as an analysis of several parallel algorithms for the single source shortest path (SSSP) problem are presented.

[1] The present study deals only with arcs with non-negative weights, i.e., $w : E \to \mathbb{R}^{*+}$

7.1.1 Point to Point (P2P)

The search for the shortest path between two distinct points of a graph is known as the "Point to Point" (or P2P) problem. Formally it is defined as: given a strongly connected[2] graph G, oriented or not, with weighted edges and two distinct vertices, respectively source s and target t, find all the shortest paths from s to t.

This problem can be solved by a modified version of the Dijkstra's algorithm [75], where the search ends when the target vertex is reached. This, however, does not alter the time complexity of Dijkstra in the worst case, closely related to how its priority queue is managed, ranging from the use of:

- highly expensive structures (like unordered arrays) to
- efficient structures (like binary heaps [138] and Fibonacci heaps [97]).

It is worth noting that, even with the use of the best data structures, in many real applications the shortest path algorithm must be executed a significant number of times and on large graphs, which makes its use impractical if techniques for its acceleration are not employed.

Such acceleration techniques can be used individually or in combination. Details about some of them can be found in [9, 58, 185, 101]. Regardless of how they are used, they all have one thing in common: divide the original problem in two stages, named "preprocessing" and "search".

The preprocessing stage receives a directed graph G, as previously defined, and produces some auxiliary data structure that aims to make the next stage (search) more efficient.

The content of the auxiliary data produced is heavily dependent on the method used. It can range from the use of information about the graph geometric structure, decomposition based on some hierarchy (also known as *graph partitioning*), landmark distances or the modification of the underlying graph structure [101, 108, 215, 245].

Even though all of these approaches have their merits, they also have drawbacks that limit their use. For example, in the partitioning strategy the graph is divided into multiple "chunks" of approximately the same size in a way that there are few interactions (edges) between them. The main difficulty here is that the partitioning problem belongs to the NP-Complete class, which requires the use of heuristic methods [202, 249].

On the other hand, restrictions about the time spent on the preprocessing phase depend on the application itself. If the graph is static (or rarely changes), more time can be spent in this stage, otherwise there will be a significant restriction on its utilization. A

[2]The formal definition talks about a connected graph, but this work sticks to the case of strongly connected graphs. That is, those in which there is at least one path between any two distinct vertices.

classic study in the field [188, p.-40–41], [224] points out that the preprocessing can take several hours for a quite simple problem of handling a railway network operating schedule and, therefore, it is only performed twice a year, since the structure of the network in question varies only every winter/summer. Hence, its use is justified in this context.

The search phase is the traditional processing applied to the data structure produced in the preceding step, and identifies the solution.

However, in the PET-GYN software, one of the primary goals is to propose changes to the structure of the road network and then to evaluate its consequences, i.e., to solve a Network Design Problem (NDP). This activity is performed very often by traffic engineers in their planning activities and a preprocessing stage that consumes several hours makes its use impractical in real scenarios.

7.1.2 Single Source

The two best-known algorithms for solving this problem were proposed by Bellman-Ford [12] and Dijkstra [75]. The former is more generic, since it is able of handling graphs with negative edge weights, but it is scarcely used because of its high asymptotic computational complexity, $O(n \cdot m)$. On the other hand, Dijkstra's algorithm is only applicable on graphs with no negative edge weights, but it has a smaller asymptotic time complexity.

Although Djikstra's algorithm belongs to the class of "greedy algorithms", it is guaranteed to always find the optimal solution. This is possible thanks to the use of a contrivance: during the computation, it maintains and uses a set of vertices, called *priority queue*, to guide the greedy search toward the optimal solution.

Defining s as a root vertex, Dijkstra's algorithm grows a *shortest path tree* holding the *tentative distances* (i.e., the smallest so far) to all other vertices. At each step, the vertex with the smallest distance to s is dequeued and marked as already processed (or settled), that is, its minimum distance to s has been found and, therefore, it doesn't need to be reprocessed.

As briefly mentioned in Section 7.1.1, the algorithm runtime is intrinsically linked to the way the priority queue is handled, since at every step of shortest path building, it must be scanned in order to locate the element with the smallest distance. If the queue is kept sorted by priority (distance), the cost of this location and removal is greatly reduced. However, it is important to note that the process of keeping it ordered also incurs in a certain cost. The cost of these two processes (locate/extract the minimum and keep the queue ordered) is what must be minimized, which is achieved through the use of efficient data structures. Table 7.1 lists the time complexity for several proposals [150].

Here, the *total* column depicts the time complexity of Dijkstra's algorithm according to the used data structure.

Data structure	insert	delete-min	find-min	decrease-key	total
Unordered array	$O(1)$	$O(n)$	$O(n)$	$O(1)$	$O(n^2)$
Binary heap [138]	$O(\log n)$	$O(\log n)$	$O(1)$	$O(\log n)$	$O((n+m)\cdot\log n)$
Binomial heap [244]	$O(1)$	$O(\log n)$	$O(1)$	$O(\log n)$	$O((n+m)\cdot\log n)$
d-ary heap [139]	$O(log_d(n))$	$O(d\cdot\log_d n)$	$O(d\cdot\log_d n)$	$O(\log_d n)$	$O(m\cdot\log_{m/n} n)$
Fibonacci heap [97]	$O(1)$	$O(\log n)$	$O(1)$	$O(1)$	$O(n\cdot\log n+m)$
Strict Fibonacci heap [28]	$O(1)$	$O(\log n)$	$O(1)$	$O(1)$	$O(n\cdot\log n+m)$
Relaxed heap [76]	$O(1)$	$O(\log n)$	$O(1)$	$O(1)$	$O(n\cdot\log n+m)$
Brodal queue [27]	$O(1)$	$O(\log n)$	$O(1)$	$O(1)$	$O(n\cdot\log n+m)$
Pairing queue [134]	$O(1)$	$O(\log n)$	$O(1)$	$O(\log n)$	$O(n\cdot\log n+m)$
Rank-Pairing queue [120]	$O(1)$	$O(\log n)$	$O(1)$	$O(1)$	$O(n\cdot\log n+m)$

Table 7.1: *Sequential Dijkstra: priority queue management operation costs.*

In relation to how a graph G can be stored, the best data structure depends on the problem in focus. If one is dealing with not so large graphs, an adjacency matrix may be suitable, despite its $O(n\cdot n)$ memory consumption. The checking for the presence or the absence of a specific edge can be done in constant time $(O(1))$, but to iterate over all edges is a slow process. Adjacency lists, on the other hand, only use memory in proportion to the number of vertices and edges, or $O(n+m)$ [51], which potentially can save much of memory space if the graph is sparse[3]. It is fast to iterate over all edges but finding the presence or absence of a specific edge – which can be done in $O(\log(\Delta))$ using a binary search if the adjacency list is ordered, where Δ represents the maximum degree of G – is slightly slower than with the adjacency matrix.

7.1.3 Many to Many and All Pairs

Here, the problem can be described as: given two sets of vertices $S,T\subset V$ compute the shortest path between all pairs of vertices $(s,t)\in S\times T$. It is called "Many to Many Shortest Path Problem" and, if $S=T=V$, then the problem is known as "All Pairs Shortest Paths" (or APSP).

[3] In graph theory, the distinction between dense and sparse graphs is quite vague, varying with the author and with the context. Roughly speaking, a graph can be defined as dense if the number of edges is close to its maximum value and sparse if it has few edges. West [249] says that a graph is sparse if $m\leq\Theta(n^k)$, where $1<k<2$. Lee and Streinu [159] defines that a graph is $(k,l)-sparse$ if every subset of $n'\leq n$ vertices spans at most $k\cdot n'-l$ edges. Nešetril and Mendez [186] defined two classes of dense graphs: the ones for which there exists a threshold $\boldsymbol{t}$ such that every complete graph appears as a $\boldsymbol{t}$-subdivision in a subgraph of a graph belongs to the ***somewhere dense*** class. Otherwise, if such a threshold does not exist, the class is ***nowhere dense***.

One way to solve it is simply running Djikstra's algorithm for all vertices in S, as demonstrated by [100, 191, 202]. However, there are proposals for its direct resolution [88, 99, 152, 216].

Limitations

Sometimes, when solving this kind of problem, it is not necessary to build the set of shortest paths itself, but just keeping the distances between the vertices. In this case, only one matrix that stores the distances will be needed.

However, in cases when both pieces of information (distances and shortest paths) are required, this leads inevitably to the use of two matrices: one to represent the calculated distances between all pairs of vertices of G and another to hold the predecessors of each vertex in every path $(s,t) \in S \times T$. In the case of real road networks, which are highly sparse in general, this results not only in a huge waste of memory (to store the graph) but also the impossibility of its use if the cardinality of the set V is large, because the space occupied in memory would be expressed by the double of that cardinality (for distances and predecessors).

Since the aim of the algorithm presented in Chapter 8 is to describe an efficient way to run macroscopic traffic assignment in very large networks using GPUs, the "Many to Many" and the "All Pairs" approaches cannot be applied due to its huge memory consumption.

Just as an example to illustrate the problem, let's take the California road network, which has a total of 1965206 vertices. Every vertex identifier – used in the predecessors matrix – will need 4 bytes for its storage, and another 4 bytes will be needed to hold an edge weight, used in the distances matrix. This will lead to a memory usage of $2 \cdot 4 \cdot 1965206 \cdot 1965206 = 30,896,276,979,488$ bytes, or ≈ 31 petabytes only to handle these two matrix data structures, which may be considered impractical.

Thereby, it is necessary to look for alternatives that do not use this huge amount of memory. A workaround would be to perform n SSSP separate calculations, which can be achieved using the algorithms described in Section 7.1.2, and that consume a smaller amount of memory $(O(n))$ to store the results in each execution[4]. The results can then be saved in another memory (in the main CPU RAM or in secondary storage) and the space already allocated on the GPU may be reused for the new calculations.

In fact, this alternative is very interesting and can be effectively used in traffic assignment algorithms, as described further in the thesis.

[4]For simplicity reasons, the present work is not considering for analysis the size of the problem input, but only the size of the result.

Another aspect that is worth mentioning is that, in general, when dealing with the urban Traffic Assignment Problem (TAP) it is often not necessary to know the shortest path between all pairs of vertices, just between some points of the road network defined by sets of source (S) and destination (T) vertices, with $|S| \cdot |T| \ll n^2$. Thus, the idea of only performing $|S|$ SSSP calculations appears as a natural solution to this problem. As it will be shown later, other TAP characteristics make this process even easier and reinforce the use of SSSP algorithms.

7.1.4 Classic Algorithms for the SSSP Problem

The standard SSSP algorithms are based on a *labeling* process and can be categorized into two groups according to the way the tentative distance is updated: *label-setting* and *label-correcting*, where throughout successive iterations a shortest path tree is built and improved until no further improvements are possible [255].

The methods based on the label-setting determine, in each iteration, a permanent (not subject to changes) distance label for only one vertex at a time. The label-correcting methods, on the other hand, change the distance label of every vertex several times and only after the final step they all become permanent [179]. For a good explanation about both groups, please refer to the works of Bogdanov and Trevisan [21] and Zhan and Noon [255].

The best known algorithm that uses the label-setting strategy is the one proposed by Dijkstra. Assuming that v_{sf} is the current vertex in the search frontier, at each step the method applies the *scanning* operation[5] to all vertices neighboring v_{sf} until either the target vertex is reached and defined as settled (for the P2P problem) or until all vertices have been scanned and labeled as permanent (the SSSP variant).

The Standard Dijkstra Algorithm

The standard Dijkstra approach (pseudo code shown in Algorithm 7.1) is based on an iterative *labeling process* (see Section 7.1.5). For each vertex $v \in V$, it maintains a *tentative distance label* $\varphi(v)$ and a *minimum distance* $d(v)$, where $\varphi(v)$ represents an upper bound on $d(v)$ and refers to the value of the smallest path from the starting vertex s to v found so far. As can be seen in lines 1 to 4 of Algorithm 7.1, it initially sets $\varphi(v) \leftarrow \infty, \forall\, v \neq s$ and $\varphi(s) \leftarrow 0$.

As pointed out in Section 7.1.2, the algorithm also maintains other two sets of vertices: $Q = \{v_1, v_2, \ldots, v_k\}$, $k \leq |V|$, used to guide the greedy search toward the optimal

[5] Also known as *relaxing* process, where the algorithm verifies if the tentative distance to the vertices neighboring v_{sf} can be improved.

solution, called *priority queue*, and $\pi(v)$, storing the predecessor of v in the shortest path. Initially, the only element in Q is s (line 5) and $\pi(v) \leftarrow \nexists, \forall\, v \in V$ (line 3).

In the next steps, the labeling process repeatedly selects and extracts a vertex "v_{sf}" from Q where $\varphi(v_{sf}) \leq \varphi(v_q), \forall\, v_q \in Q$ (lines 6 to 8) and v_{sf} is then marked as *settled*, i.e. its minimum distance to s has been found and, therefore, $d(v_{sf}) \leftarrow \varphi(v_{sf})$. Then, vertex v_{sf} becomes the new *search frontier*.

Finally, each $v_a \in Adj(v_{sf})$ is checked to verify if $d(v_{sf}) + w(v_{sf}, v_a) < \varphi(v_a)$ (lines 6 to 8). If the condition holds, then $\varphi(v_a) \leftarrow d(v_{sf}) + w(v_{sf}, v_a)$ and $\pi(v_a) \leftarrow v_{sf}$. In the last step (line 14), the algorithm inserts v_a in Q.

During the labeling process, each vertex $v \in V$ may be in one of the following ***states***:

- *unreached*: the vertex has not been inserted in Q and, therefore, $\varphi(v) = \infty$;
- *labeled*: the vertex was checked at least once during the neighbor relaxing process and now belongs to Q. It can also be called *queued* or *candidate*;
- *settled*: the minimum distance from s to v has been found and v was removed from Q.

A direct consequence of dealing only with non-negative weighted edges is that every vertex visited during the relaxing process will be in a non-decreasing distance from its predecessor vertex. Hence, the smallest Dijkstra's time complexity ($O(n \cdot \log n + m)$) presented in Table 7.1 is the best possible upper-bound when methods based on comparison of values are employed.

This happens because any hypothetic SSSP algorithm with $O(n \cdot \log n)$ time will violate the $\Omega(n \cdot \log n)$ lower-bound for comparison-based sorting methods [51, 175][6].

[6] Although this is true for generic graphs, there are some classes of graphs (like planar [237] or with separator decomposition [48]) for which more efficient algorithms are known. In [233] and [234], Thorup presents two algorithms for solving the SSSP problem with $O(n+m)$ time and space complexity for the special case of undirected graphs with integer weights. Unlike Dijkstra, which visits vertices in the order of increasing distance, his proposal is to traverse a *component tree*. Algorithms with linear average time for uniformly distributed edge weights were presented in [107, 176].

Algorithm 7.1: *DijkstraAlgorithm(G, s)*

Input: Non-negative weighted connected simple graph ***G = (V, E)***, a starting vertex ***s*** and a set $w : E \rightarrow \mathbb{R}^+$ of edge weights.

Output: Two sets: φ (distances) and π (predecessors).

```
1  foreach v ∈ V(G) do
2      φ(v) ← ∞;
3      π(v) ← ♯;
4  φ(s) ← 0;
5  Q ← s;
6  while Q ≠ ∅ do
7      u ← extractMin(Q);
8      Q ← Q − {u};
9      foreach v_a ∈ Adj(u) do
10         dist ← φ(u) + w(u, v_a);
11         if dist < φ(v_a) then
12             φ(v_a) ← dist;
13             π(v_a) ← u;
14             Q ← Q ∪ {v_a};
```

The Standard Bellman-Ford-Moore Algorithm

The Bellman-Ford method is a well-known label-correcting algorithm for the SSSP problem, and allows, as aforementioned, edges with negative weights. A curious fact about it is that around the same time a different researcher, Edward Moore, published an equivalent strategy in another article, and for this reason it is also known as Bellman-Ford-Moore (BFM) algorithm.

Bellman-Ford-Moore's approach is based on an iterative *label-correcting* process shown in Algorithm 7.2. It performs a linear number of repetitions (line 5) over the entire input graph (line 7), therefore ending in a polynomial time of $O(n \cdot m)$.

Its basic implementation has some similarities with Dijkstra's algorithm. During the initialization process, it sets $\varphi(v) \leftarrow \infty, \forall\, v \neq s$ and $\varphi(s) \leftarrow 0$. In the relaxing process, it updates sets $\varphi(v)$, $d(v)$ and $\pi(v)$ in an analogous way.

After initilization, in the relaxing process (lines 7 – 12), all edges $(v,u) \in E$ are verified to check if the constraint $d(u) \leq d(v) + w(v,u)$ is respected. If it is violated, then $d(u) \leftarrow d(v) + w(v,u)$ and $\pi(u) \leftarrow v$. The process repeats $|V|$ times or until no edge has been relaxed. In any case, when the algorithm ends it will have either solved the problem or will have a proof that the problem has no solution (lines 15 – 17).

Algorithm 7.2: *BellmanFordMooreAlgorithm(G, s)*

Input: Weighted connected simple graph ***G = (V, E)***, a starting vertex ***s*** and a set $w : E \to \mathbb{R}$ of edge weights.

Output: Two sets: ***d*** (distances) and π (predecessors).

```
1  foreach v ∈ V(G) do
2      d(v) ← ∞;
3      π(v) ← ♯;
4  d(s) ← 0;
5  for i ← 1 to |V(G)| do
6      relaxed ← false;
7      foreach (v,u) ∈ E(G) do
8          dist ← d(v) + w(v,u);
9          if dist < d(u) then
10             d(u) ← dist;
11             π(u) ← v;
12             relaxed ← true;
13     if relaxed = false then
14         exit the loop;
15 foreach (v,u) ∈ E(G) do
16     if d(u) > d(v) + w(v,u) then
17         return false;
18 return true;
```

7.1.5 Parallel Algorithms for the SSSP Problem

Researchers in the SPP area have continuously strived to develop more efficient strategies than those already created. The literature is abundant in algorithmic solutions to this problem. Although the existing proposals are very different among themselves, at the end, they can be classified into three main groups:

1. The ones that identify new properties or structures in the graph and in the problem that allow building methods with smaller asymptotic complexity;
2. The ones that compress the graph in order to reduce the size of the problem instance; and
3. Those that explore computational parallelism.

We focus now on the third group. Furthermore, there are two main approaches to implement parallelization in the standard SSSP algorithms. The first one parallelizes

the intrinsic operations performed by the algorithm itself. The second approach splits the graph in sub-graphs and simultaneously applies the sequential algorithm to each sub-graph [231, 228]. In this thesis, only the first approach was investigated.

Regarding suitability for parallelization, the two most well-known sequential algorithms for SSSP (see Section 7.1.4) have their drawbacks. The standard Dijkstra, despite its low complexity, has an inherent sequential nature, since only one vertex is processed in each iteration. On the other hand, BFM allows a more direct parallel approach since it processes all edges in each iteration, repeatedly updating the vertices' smallest distance until the final distances are found. But this has a high computational cost.

All strategies presented next try to, somehow, overcome these limitations. They are applied to directed graphs and have their strengths and weaknesses, which are summarized in Table 7.2. Some evaluations of parallel methods for *undirected* graphs are available in [228]. A more comprehensive and detailed listing of parallel algorithms for the SSSP is available in [174].

Crauser et al.

In the strategy depicted in [53], the authors split Dijkstra's sequential approach into a number of *phases*, such that the operations within a phase can be done in parallel. Parallelism is achieved by extracting of more than one vertex from the priority queue Q at each iteration and relaxing their outgoing edges also in parallel. The difficulty lies in how to identify the set of vertices that can be simultaneously removed without affecting the algorithm's correctness. They present some criteria for how to construct such a set:

- The first criteria, called *OUT-version*, finds a threshold L with the weights of the *outgoing* edges, defined as $L = \min_{\forall\, v \in Q} \{\varphi(v) + w(v,u) | (v,u) \in Adj(v)\}$. Using this threshold, the algorithm defines as settled and removes from Q all vertices with $\varphi(v) \leq L$. It also relaxes all their outgoing edges;
- The second criteria, called *IN-version*, computes two thresholds: $M = \min_{\forall\, v \in Q} \{\varphi(v)\}$ and another one with the weights of the *incoming* edges, defined as $i(v) = \{\varphi(v) - \min\{w(u,v), (u,v) \in Inc(v)\}\}$. All vertices in Q that satisfy the condition $i(v) \leq M$ can be safely defined as settled and removed from Q. As in the first criteria, the algorithm also relaxes all their outgoing edges;
- Finally, the *IN-OUT-version* applies both criteria in conjunction.

In their implementation, a global array is employed to maintain $\varphi(v)$, $\forall\, v \in V$, i.e., the tentative distance of all vertices. Every processing unit handles a subset $S \subset V$ of randomly assigned vertices and has two sequential priority queues.

The first priority queue stores $\varphi(v)$, for $\forall\, v \in S$, and the second one stores the addition of $\varphi(v)$ and its minimum outgoing edge weight. In order to accelerate this step, in a pre-processing phase the outgoing edge with the minimum weight for every vertex is determined.

Next, the authors show how these variants can be efficiently implemented on an arbitrary-write CRCW PRAM. The tests were performed on random directed graphs under the model $G(n, \frac{d}{n})$, where $\boldsymbol{n}$ represents the number of vertices in the graph and each possible edge is included with probability $\frac{n}{d}$. Furthermore, the edge weights were uniformly distributed in $[0, 1]$.

The performed experiments showed that the OUT-version was able to find the solution in $2.5\sqrt{n}$ phases. A refined IN-OUT-version (using an alternative implementation based on a parallel priority queue) needs about $6.0\sqrt[3]{n}$ phases on average. Other experiments were performed and presented varying levels of optimization, according to the used version and the graph type.

Brodal et al.

As shown in Section 7.1.2, the running time of Dijkstra's algorithm is intrinsically linked to the way the priority queue Q is managed and the use of adequate data structures can significantly improve the performance.

In the works presented in [29, 30], Brodal et al. show two different ways to efficiently handle the elements of Q in parallel. The first way speeds up the queue operations that deal with a single element in Q using a small, limited number of processing units. The second method adds to Q the support of simultaneous vertices insertion and simultaneous removal of the smallest elements (the vertices with smallest distances). In both cases, a data structure representing a parallel priority queue able to perform its internal operations (insert, update, etc.) in $O(1)$ time is employed.

A new parallel alternative to Dijkstra's algorithm is presented using this data structure. In their implementation, they represent the graphs as adjacency lists and keep these lists sorted using their weights, making possible in constant time both the determination of the vertex with minimum distance and the addition (in parallel) of an arbitrary amount of vertices to the data structure. The update distance operation can also be performed in parallel.

To support this data structure, a processor pipeline is employed, where each processing unit receives the data produced by its predecessor, performs a constant time merge operation and selects the data to be sent to the next processing element. They also defined that each vertex $v \in V$ has a dedicated processing element.

In their implementation, two sets that make use of this data structure have been defined: the first one, S, represents the list of vertices already defined as settled (i.e.,

those ones whose shortest path have been found), and $S' = \{\{Adj(v), \forall\, v \in S\} - S\}$ is a collection of neighbors of vertices in S, excluding the ones in S.

Besides that, among the processing elements assigned to vertices in S, one will randomly be elected as a master processor. The algorithm will not finish until the master processor determines that the priority queue is empty.

Four operations are defined for this data structure:

- *INIT* – initialization the data structure;
- *EJECT* – removal of the element with minimum weight from S and sending it to the master processor;
- *EXTEND* – consists of assigning a fixed weight to a vertex and adding it to S. The processor assigned to this vertex becomes the new master processor;
- *EMPTY* – only performed by the master processor, consists of checking the emptyness of S.

The authors claim that, under their technique and if $O(\frac{m \cdot \log n}{n})$ processors are available, Dijkstra's algorithm can be implemented to run in $O(n)$ time and $O(m \cdot \log n)$ work complexity on a CREW PRAM.

Martín et al.

The idea behind the parallelization method proposed by Martín et al. [171] is that the standard Dijkstra implementation only deals with a unique vertex v_{sf} in the search frontier, even when several tentative distances $\varphi(v)$ in Q coincide with the current minimum. When this happens, the algorithm randomly chooses one of them to compose the new frontier. As a consequence, it will take several iterations to settle each one up and remove it from Q.

Therefore, instead of having a *simple* frontier, in the proposed implementation they design the *Dijkstra's Algorithm Adapted to* ***Compound*** *Frontiers (DA2CF)*, able to handle a set F of frontier vertices. Although in this new approach the same three basic operations of the standard version are performed, they need to be adapted in order to handle the compound frontiers:

- *Relax:* Consists of updating the shortest path estimate for all vertices in Q using the elements in F. Therefore, the value $\varphi(u) \leftarrow \min\{\varphi(u), d(v) + w(v,u),\ \forall\ (v \in F, u \in Q)\}$ must be computed, which can be performed in parallel if a processing element is assigned to each $(v \in F, u \in Q)$. However, it should be noted that this can lead to inconsistencies if two vertices $v_1, v_2 \in F$ concurrently try to update the same

vertex $u \in Q$. In order to prevent this, in their implementation the authors adopt an *atomicMin* instruction[7], which may potentially serialize the whole operation;
- *Minimum:* involves finding the minimum estimate value (called *mssp*) in Q. In this step they used an adapted version of the *reduce3* method described in [124] (see Chapter 6 for further details);
- *Update:* it updates the set Q, removing vertices wit distance estimate $\varphi(u) = mssp$. The removed vertices will form the new set F.

Arranz et al.

The work presented in [190] by Arranz et al. adapts the Crauser's algorithm [53] to GPU architectures and performs experimental comparisons with both CPU and GPU implementations of Martín et al. [171].

In their implementation, two types of graph representations were employed: adjacency lists and matrices. Besides the basic structures to hold nodes, edges and the respective weights, three other vectors were defined:

- U – Stores in $U[v]$ whether the node v belongs to the unsettled set;
- F – Defines if $F[v]$ is a node in the frontier set;
- δ – Stores in $\delta[v]$ the tentative distance from the source to node v.

Arranz et al. performed experiments evidentiating speedups varying from 13x to 220x when compared to CPU times and a performance improvement up to 17% with respect to the GPU-Martín et al. algorithms. Among its drawbacks, the following ones can be cited:

- The best speedups were achieved using the matrix representation, which precludes the use of this method in larger graphs;
- The sets U and F are entirely allocated in global memory, which ultimately leads to a misaligned access pattern during the relaxation step, downgrading the desired speedup;
- Each *processor* relaxes all neighbors of its starting vertex. If the vertices have a very irregular outdegree distribution, this will induce an irregular workload for each *work-item* in the relaxation step; and

[7] Ultimately, the employment of this atomic instruction prevents its use if two sets of values ($\varphi(v)$ and $\pi(v)$, see Section 7.1.4) need to be updated in the step of computation. Moreover, there are only atomic instructions in hardware for simple types like integers. Its not possible to implement atomic instructions in hardware to more complex types of data, such as floats and, if these instructions are needed, they must be emulated via software.

- Since the expanded search frontier manages more than one vertex at the same time, during the relaxation step two or more *work-items* can relax the same neighboring vertex simultaneously. Therefore, to ensure that the minimum tentative distance is written, the use of an *atomicMin* is required, serializing the process.

Meyer & Sanders

In [174], Meyer & Sanders propose a parallel version of a label-correcting algorithm for the SSSP problem called Δ-stepping, where an ordered list of eligible vertices with their respective tentative distances is held in a collection of *buckets*, representing priority ranges of size Δ (the bucket width), and where each element in the bucket can be processed in parallel.

The basic supporting idea of the presented algorithm is a weakening in the total ordering of the elements in the priority queue Q, only employing an array B of *buckets*[8] such that $B[i]$ maintains just the vertices $\{v \in V, v \text{ is queued and } \varphi(v) \in [i \cdot \Delta, (i+1) \cdot \Delta]\}$, in the i^{th} iteration (or phase).

Using the Δ parameter, they also introduced the concepts of *light* and *heavy* edges, where the light edges are the ones where the condition $\{(v,u) \in E : w(v,u) \leq \Delta\}$ holds and the heavy edges is the set where the condition $\{(v,u) \in E : w(v,u) > \Delta\}$ is satisfied.

In each phase, every vertex in $B[i]$ is removed and all *light* outgoing edges are relaxed, potentially introducing new vertices in $B[i]$. It is worth note that if a vertex v is removed from the current bucket without its definitive distance, in some next step of the same phase it will be surely reintroduced in $B[i]$, which ensures the correct computation of its minimum distance – in fact, if $\Delta = \infty$, then the proposed algorithm will become the standard Bellman-Ford.

The remaining set of *heavy* edges is entirely relaxed only once, when the current bucket gets empty, what assures that their corresponding starting vertices are marked as settled and that the minimum distance to each $v \in B[i]$ was found.

Under this strategy, the parallelism is now straightforward, by simultaneously removing all vertices in $B[i]$, relaxing their outgoing edges where $\{(v \in B[i], u \in Adj(v)) \in E : w(v,u) \leq \Delta\}$ and, finally, relaxing all heavy edges where $w(v,u) \in E > \Delta$.

As pointed in the conducted experiments, the performance of the presented strategy is strongly dependent on the choice of the value of Δ, which offers a trade-off between too many node reintroductions in the current bucket and too many bucket traversals [228]. The challenge here, therefore, is to find a value of Δ that fits well between these two extremes.

[8] In the conducted experiments, buckets were implemented as doubly linked lists.

Papaefthymiou & Rodrigue

In [194], Papaefthymiou & Rodrigue explore the fact that, under the Bellman-Ford-Moore approach, edges can be relaxed in an arbitrary order without affecting the algorithm correctness. In their parallel strategy, they do not adopt a dynamic load balancing of the data; instead, all the information to be processed is distributed once (in a highly uniform way) to the processors at the beginning of the computation.

For each vertex $v \in V$, the partitioning process tries to keep all neighboring vertices $u \in Adj(v)$ on the same processor. Data is distributed into $\boldsymbol{P}$ chunks of roughly the same size – where $\boldsymbol{P}$ is the number of processor – and assigning a chunk to each processor.

The experiments performed showed that the proposed algorithm achieves better results on dense graphs than on sparse ones, specially when the proportion $\frac{E}{V}$ exceeds 2^5 or 2^6. For extremely sparse graphs, where $E \approx V$, the algorithm performance was poor, probably due to the little work processors could perform between the synchronization/-communication steps.

Hajela & Pandey

In [121] Hajela & Pandey propose two parallel versions of the BFM algorithm, one to compute the SSSP and a modified version to solve the APSP problem.

As pointed in their work, since the k^{th} value of $d(v)$ depends on the value computed in the k^{th-1} iteration, its not possible to get rid of (parallelize) the outer loop (line 5 of Algorithm 7.2). Therefore, all possible parallelism relies in the relaxing process (lines 7 – 12), where two levels are possible:

1. In the k^{th} iteration, the values of $d(v_1)$ and $d(v_2)$ do not depend on each other for any $v_1, v_2 \in V$;
2. For all $v \in V$, $d^{th-1}(v) + w(v,u)$ can be computed in parallel.

Their strategy is highly dependent on the matrix representation of the graph which, ultimately, prevents its application to larger datasets (see Section 7.1.3). Despite this limitation, the experiments performed by Hajela & Pandey on random graphs showed speedups varying from 13.8x to 18.5x, when compared to the standard sequential version.

Jeong et al.

The GPU parallel strategy for BFM presented in [136] by Jeong et al. launches multiple threads, one for each $e \in E$, and concurrently updates the shortest path information instead of sequentialy computing every minimum distance. For this, a GPU *kernel* repeatedly relaxes the associated edge until the shortest paths are found.

In the experiments performed, the authors did not provide any significant information about the graphs used in the experiments, only the number of vertices of each one. An implementation made by the author of this thesis, strictly following the directions provided in the article, did not confirm the alleged results.

Agarwal & Dutta

In [2] Agarwal & Dutta present two GPU parallel algorithms based on BFM. The first one is quite similar to strategy presented in [136], where in $|V|$ iterations $|E|$ threads are launched, one for every edge $e \in |E|$, and relax their assigned edge. Since this update process can potentially lead to race conditions[9], to ensure its correctness an *atomicMin* is employed.

The second one, called *Parallel BFM using Two Flags*, uses two vectors of flags, F_1 and F_2, in order to relax only those edges which source node was updated in the last iteration. This reduces the computation time since, at each iteration, just a subset $e' \in |E|$ needes to be updated.

Both algorithms were implemented by the author of this thesis. Using the second strategy, the performed experiments showed some improvements over the sequential BFM and the first parallel implementation. However, it is worth to note that even this strategy does not beat the standard Dijkstra algorithm, being indicated only in cases where Dijkstra can not be applied.

Kumar et al.

In [157], Kumar et al. present a modified version of the BFM algorithm using CUDA that performs well in dense graphs. In the proposed strategy, after each execution of the outer loop in line 5 of Algorithm 7.2, it is verified whether a solution was already found and, therefore, the outer loop doest not necessarily have to iteract $|V|$ times.

In the performed experiments, when the strategy is applied to very dense graphs it takes no more than 20 iterations of the outer loop to find the problem solution, due to this low diameter property. For graphs that take exactly $|V|$ iterations for SSSP computation, their algorithm will not get any speedup when compared to the standard implementation.

[9] A ***race condition*** occurs when code running on two or more hardware devices (processors or I/O elements) have access to shared data and some of them try to write to the same location at the same time. Since the scheduling algorithm (either software or hardware implemented) can swap between the running code at any order and time, it becomes impossible to know the order in which the shared data will be changed. Therefore, the result of the write operation is heavily dependent on the scheduling algorithm. In other words, all hardware devices are "racing" to access and/or change the shared data.

7.1.6 Overview of the Strategies

All the aforementioned strategies try to explore different types of parallel mechanisms to speedup the SSSP resolution and each one has its strengths and weaknesses. Table 7.2 sumarizes their main aspects.

Algorithm	Based on	Strategy	Strengths & Weaknesses
Crauser et al.	Dijkstra	Split the algorithm into phases and solve each one in parallel.	A relatively small number of steps is required for finding the solution. On the other hand, the employed dynamic data structures and the CRCW PRAM model are not suitable for use in GPUs.
Brodal et al.	Dijkstra	Adds to Q the support of simultaneous vertices insertion and removal of the smallest elements.	Execution in linear time in a CREW PRAM machine. But, as in Crauser et al., the employed dynamic data structures are not suitable for use in GPUs.
Martín et al.	Dijkstra	The expanded frontier is able to handle a set of vertices.	A real GPU implementation. Among its drawbacks, the usage of atomic operations in the update process and the edge weitghts limited to integer values.
Arranz et al.	Dijkstra	The expanded frontier is able to handle a set of vertices.	A real GPU implementation of Crauser et al. Converge to a solution in a relatively small number of iterations. The bottleneck lies in the constant communication between CPU and GPU, besides that only distances are computed, ignoring the predecessors.
Meyer & Sanders	BFM	An ordered list of eligible vertices is held in a collection of buckets where each element can be processed in parallel.	Good performace for random graphs with random edge weights. Dynamic data structures and the PRAM model are not suitable for use in GPUs. Its performance is heavily dependent on the value of Δ.
Papaefthymiou & Rodrigue	BFM	Explores the fact that edges can be relaxed in an arbitrary order without affecting the algorithm correctness.	Achieves good results on dense graphs. For extremely sparse graphs the algorithm performance is pretty bad.
Hajela & Pandey	BFM	Strategy based on the matrix representation of the graph.	Speedups varying from 13.8x to 18.5x, when compared to the standard sequential version. Strategy highly dependent on the matrix representation of the graph, preventing its application to larger datasets.
Jeong et al.	BFM	Launch a GPU thread for each edge and repeatedly relax all vertices.	The article does not provide further details on the graphs used in the experiments, only the number of vertices on each one. An implementation made by the current author, strictly following the directions provided in the article, did not confirm the alleged results.
Agarwal & Dutta	BFM	Uses two vectors of flags in order to relax only those edges whose source node was updated in the last iteration.	GPU algorithm that achieved a certain speedup when compared to a purely sequential version. For graphs with edges with positive weights, the traditional sequential version of Dijkstra is still better.
Kumar et al.	BFM	After each execution of the outer loop, the algorithm verifies whether a solution was already found.	Achieves good results on very dense graphs. Not a real parallel approach, but just a checking that can also be applied to the sequential version.

Table 7.2: *Parallel Methods for SSSP.*

None of the aforementioned strategies efficiently meet the requirements of the system proposed in Chapter 8, which leads to the need of exploring alternative solutions to an efficient SSSP resolution. The following sections describe how this goal can be achieved using GPUs.

7.2 SSSP and its Suitability for GPU Processing in Urban Traffic Assignment Problems

As briefly mentioned in the beginning of the current chapter, a fast resolution of the shortest path problem is one of the key steps in the more complex problem of macroscopic assignment of urban traffic, described in details in Chapter 8. Since the goal of the strategy presented in that chapter is to efficiently perform the assignment process of very large networks using GPUs, all shortest paths resolution techniques that demand full matrix data structures (for holding the travel distances and predecessor vertices) cannot be considered due to their huge memory consumption (see Section 7.1.3).

SSSP appears then as a natural choice to deal with such a demand, when it is solved iteratively for each node of the network. However, none of the parallel approaches described in Section 7.1.5 solves the problem under study here well. This can be seen in the "Strengths & Weaknesses" column of Table 7.2, and involves basically the following drawbacks:

- Usage of unrealistic programming models, like PRAM, not applicable to real machines, especially considering the restrictive architecture of GPUs;
- The need of a huge amount of memory for holding the data structures, what prevents its usage for larger networks;
- Dependency of dynamic data structures, which are not suitable for GPU programming;
- Exclusive concern with shortest path cost, when the actual paths are another essential piece of information in the current work;
- Disregard for the memory hierarchy of GPUs, an essential aspect to achieve good speedups when programming such devices.

In addition to the aforementioned problems, most of the strategies presented in Section 7.1.5 reach their peak performance on dense graphs. However, since the graphs considered in this study are directed (digraphs) representing streets and intersections, they are often quite sparse and have large *diameters*[10]. Furthermore, some of the strategies deal only with integer arc costs, while the traffic assignment algorithms are based on models that consider non-negative and continuous variables represented as arc costs[11].

One should note that, although sparse, urban traffic networks are, in general, non-planar, due to the existence of bridges, tunnels, underpasses, overpasses, etc in the

[10]The diameter of a graph G, or $diam(G)$, is the largest distance $d(x,y)$ between any pair of vertices x and y in the same component of G. Formally, $diam(G) = max\{d(x,y) : x,y \in V(G),\ d(x,y) < \infty\}$. Investigations on graph theory [209] have shown that sparse graphs usually have large diameters.

[11]The cost associated to each edge usually represents the travel time or another non-negative value.

road mesh. This means that any SSSP approach that could benefit from planarity does not have a straightforward visible advantage.

All these drawbacks leave a gap that demands the development of a new parallel approach for the urban traffic assignment, as presented in the next sections. The study here focuses on aspects usually ignored by the current literature and where the other shortest-path methods did not perform well. These aspects helped to design and implement a more effective strategy for GPU parallelism. They are:

- The GPUs' memory hierarchy and how it is organized (see Sections 3.2 and 3.2.1) require an efficient partitioning of the data to be processed in order to minimize the running time of the algorithms.
- Traffic networks usually have vertices with very few neighbors (small *in/out* degrees[12]) and have large diameters. Incidently, this leads to a very interesting behavior of the evolution of the priority queue when using the traditional and sequential Dijkstra algorithm, which highlights possible ways of exploring the memory hierarchy of the GPU.

The idea for an efficient use of the memory hierarchy is described later, in Section 7.3. Firstly, the properties of urban traffic networks regarding vertex degrees and diameters are illustrated and the evolving behavior of Dijkstra's priority queue is explained.

7.2.1 A Study on Dijkstra's Priority Queue Behavior

Usually, when dealing with graphs representing urban road networks, each node corresponds to a junction (intersection) in the road mesh, and each directed edge (an arc) to its connecting element, where it can be a street, an avenue, etc.

Because of the way cities are normally built, this creates an interesting and easily observable phenomenon: all nodes of the road network have a very small number of neighbors, with the vast majority having four or fewer neighbors, which produces nearly a grid structure and leads to large diameters in the corresponding graph[13]. In mathematical terms, this means that the in/out degrees of each node are almost always very small.

Figure 7.2 illustrates this phenomenon for a small region of the city of Goiânia, capital of the state of Goiás, Brazil. The numbers highlight some intersections and

[12]The degree of a vertex v, also called *local degree* or *valency* is the number of edges connected to it. For directed graphs there are two types of degrees, known as *indegree* (the number of inward directed edges) and *outdegree* (the number of outward directed edges).

[13]Tunnels, bridges, underpasses and other structures can connect distant regions and thus reduce the diameter of the road network. However, usually a city has a much greater amount of simple components in a grid form than of those special structures, such as streets and intersections.

their respective outdegrees. Tables 7.3 to 7.5 and Figures 7.3 to 7.5 show the outdegree frequency distribution for three well-known USA road networks [160]. As can be noted, more than 99% of the nodes of the road mesh have four or less neighbors.

Figure 7.2: *Intersections and their respective outdegrees for a small region of the city of Goiânia, Goiás, Brazil.*

Outdegree	Number of nodes	Percentage
1	188317	17.30700833%
2	90740	8.33937603%
3	532686	48.95601563%
4	267256	24.56191624%
5	7759	0.71308374%
6	1237	0.11368534%
7	80	0.00735233%
8	13	0.00119475%
9	4	0.00036762%
Total	1088092	100.00%

Table 7.3: *Table of the graph outdegree distribution for the Pennsylvania network.*

Outdegree	Number of nodes	Percentage
1	251082	18,19544219%
2	115639	8,38014170%
3	699330	50,67913505%
4	307341	22,27242653%
5	5650	0,40944492%
6	808	0,05855425%
7	48	0,00347847%
8	14	0,00101455%
12	5	0,00036234%
Total	1379917	100.00%

Table 7.4: *Table of the graph outdegree distribution for the Texas network.*

Outdegree	Number of nodes	Percentage
1	321027	16,33553938%
2	204754	10,41895862%
3	971276	49,42362277%
4	454208	23,11248795%
5	11847	0,60283757%
6	1917	0,09754703%
7	143	0,00727659%
8	30	0,00152656%
9	1	0,00005089%
10	2	0,00010177%
12	1	0,00005089%
Total	1965206	100.00%

Table 7.5: *Table of the graph outdegree distribution for the California network.*

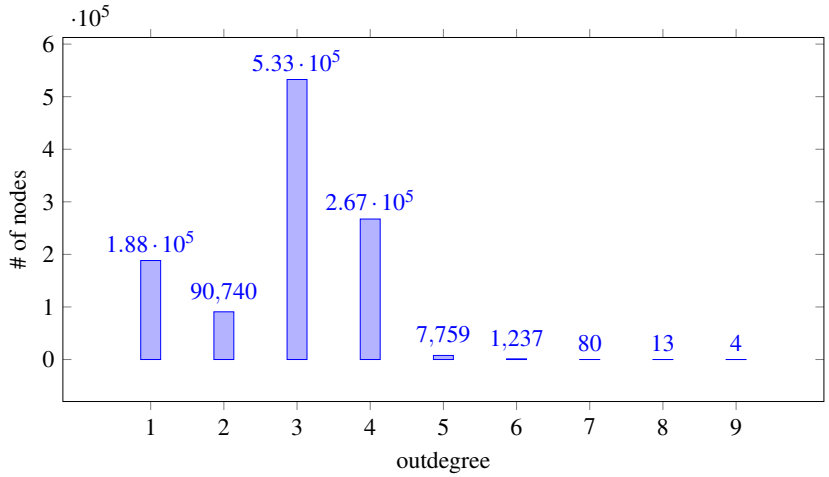


Figure 7.3: *Chart of the graph outdegree distribution for the Pennsylvania network.*

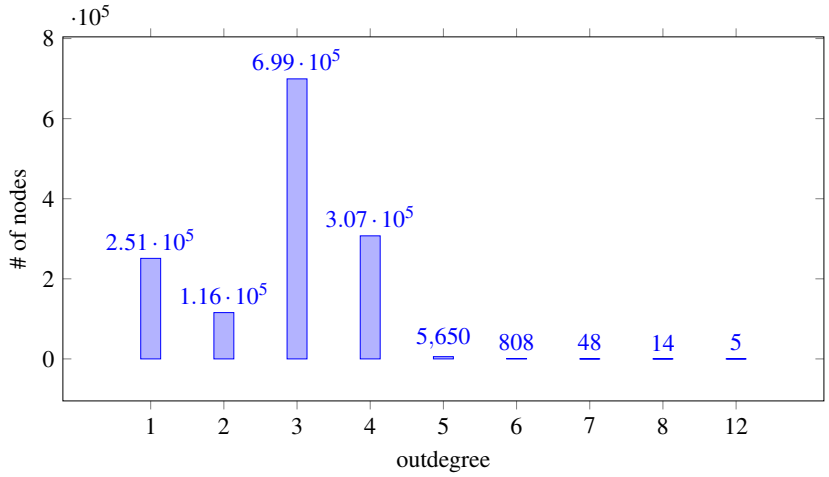


Figure 7.4: *Chart of the graph outdegree distribution for the Texas network.*

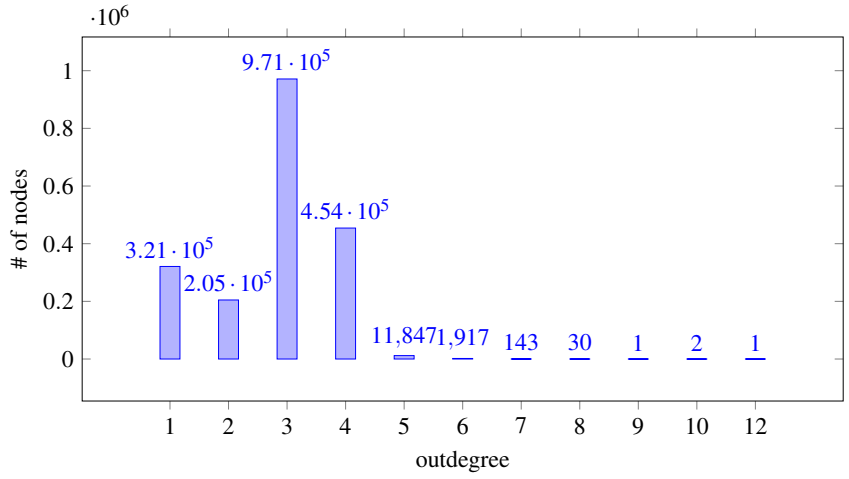


Figure 7.5: *Chart of the graph outdegree distribution for the California network.*

As a direct consequence of this phenomenon, when Dijkstra's algorithm is applied over such class of graphs, during the relaxing step only a few amount of vertices will be inserted in the priority queue Q[14]. Since, on each step, only few elements are added to Q and the vertex with smallest distance to s is always removed, this means that the number of elements that Q must hold does not grow in an explosive way, leading to a "*well-behaved*" priority queue.

Figures 7.6 to 7.8 depict the behavior of Q for the three aforementioned graphs. Despite the fact that they have a large amount of components (nodes and edges), the number of elements in Q never exceeds a tiny percentage of the total number of vertices of the graph.

This behavior, ultimately, opens the possibility of fully allocating Q in the small, but very fast, local memory. The following sections describe how this goal can be achieved.

It is worth a brief note before we continue. All these charts were produced by running the sequential Dijkstra's algorithm with the starting node equal to 0. If another starting node is chosen, the shape of the curves may be different, but it does not invalidate the analysis made here as the maximum number of elements in Q tends to be very regular.

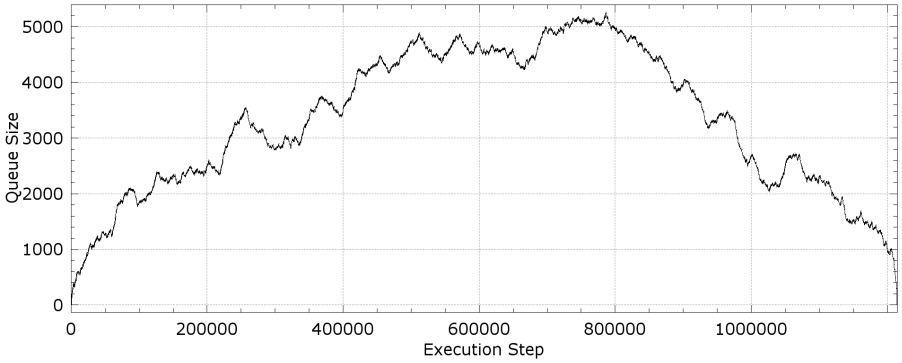


Figure 7.6: *Sequential Dijkstra: heap behavior on the graph representing the Pennsylvania network.*

[14]There is also the case where all neighbors of the current vertex don't have its smallest distance improved in the current step and, hence, Q is not expanded.

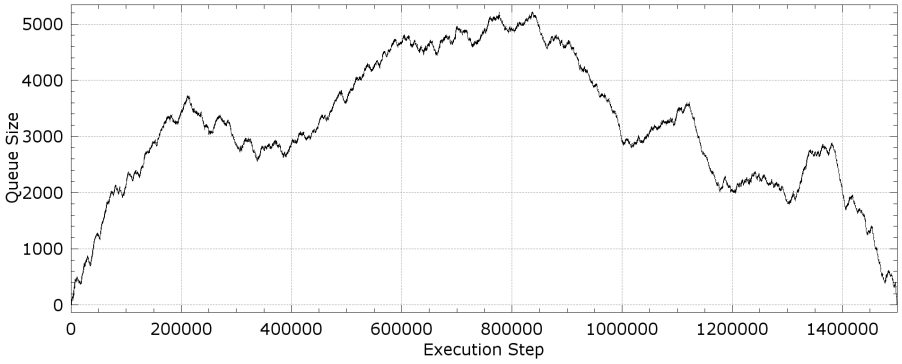


Figure 7.7: *Sequential Dijkstra: heap behavior on the graph representing the Texas network.*

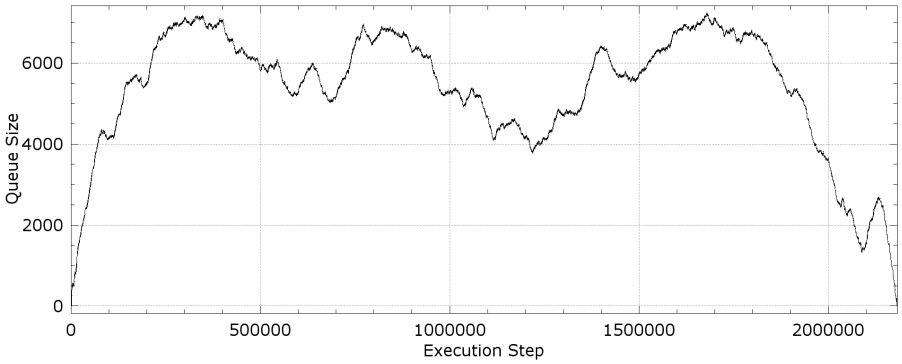


Figure 7.8: *Sequential Dijkstra: heap behavior on the graph representing the California network.*

7.3 The Proposed GPU Dijkstra Algorithm

In this section, we present our Dijkstra parallel algorithm. The approach adopted was to split the standard algorithm into three stages that are performed inside every single parallel work-item and repeated until the priority queue is empty.

The whole process starts with $|SM| * MaxSMSize$ work-items been launched by the CPU, with |SM| work-itens for each SM.

The first stage involves relaxing the neighbors of the current vertex (Lines 9 – 13 of Algorithm 7.1). The second stage takes each relaxed neighbor and inserts it in the priority queue (Line 14). The third stage locates and extracts the vertex with smallest distance from the priority queue (Line 7).

The proposed parallel algorithm was mapped to a GPU architecture. In the next section, a set of data structures needed for the maintenance of the priority queue in the local memory is presented. After that, the details of the three parallel stages are described.

7.3.1 Data Structures

As in Chapter 5, we store our graph using the compact graph representation proposed by Harish and Narayanan [123], only excluding vector L_v, which is not necessary here.

This is saved in the global GPU memory together with vectors d and π for having the tentative/definitive distances and the predecessor vertex, respectively.

For the priorty queue, referred as Q, it is important to define first what kind of information it must hold. As mentioned in Section 7.2, the traditional parallel methods for SSSP do not consider the actual paths and this information is crucial during the demand distribution step of the macroscopic traffic assignment algorithm (see Section 8.1.2).

Hence, there is the need to expand the priority queue Q in order to store not only the distance but also the node identifier, necessarily leading to the use of two data structures: φ to hold the tentative distance and V to maintain the vertex identification, both held in the local memory of each SM. So, the maximum number of elements that our expanded priority queue is able to handle is limited by the size in bytes of this local memory (ψ) divided by the cardinality of $|\varphi| + |V|$, or $|Q| = \frac{\psi}{|\varphi|+|V|}$. Figure 7.9 shows Q in the local memory.

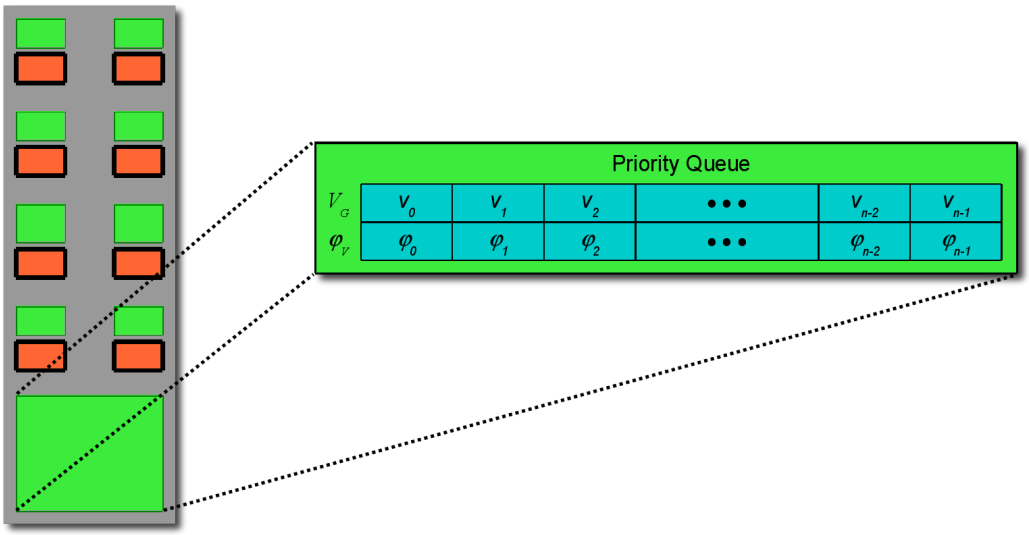


Figure 7.9: *Allocating Dijkstra's priority queue on local memory.*

The problem now is that, even if the number of elements in the priority queue does not grow significantly, as pointed out in Section 7.2.1, yet the local memory of only one SM is not able to keep all elements of Q at once. To overcome this, the strategy is to distribute the priority queue through all SMs of the GPU, where each SM maintains its own space to store "chunks" of Q, as shown in Figure 7.10.

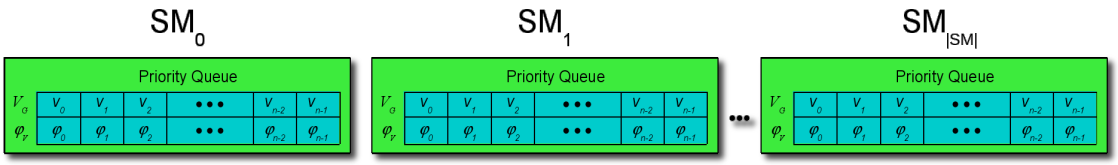


Figure 7.10: *Distributing Dijkstra's priority queue on streaming multiprocessors. All chunks of Q across the SMs have the same size, but hold different elements.*

7.3.2 First Stage

The first stage of our parallel strategy refers to the relaxation of neighboring vertices of the frontier vertex u (Lines 9 – 13 of Dijkstra's sequential version – see Algorithm 7.1).

In the first execution of the parallel algorithm, vertex u is defined as being the initial vertex s. In the other cases, this information comes from Stage 3.

For this first stage, every parallel work-item can individually analyze a distinct neighbor v_a of u and check if the current distance $d(v_a)$ from s (the starting vertex) to v_a, can be improved. This can be done by looking up $d(u)$, $d(v_a)$ and the cost W of the edge (u, v_a) in the global memory and verifying if $d(u) + W < d(v_a)$. Since we are dealing with extremely sparse graphs, only κ work-items are necessary for doing a useful work, where κ refers to the number of neighbors of u.

Furthermore, as we are working with graphs where κ is significantly smaller than the amount of work-items inside a SM, it may seem that only the work-items of a single SM are enough to analyze the neighbors of u in parallel.

However, in our strategy, the first κ work-items of **ALL** SMs simultaneously analyzes the same neighbors of u. As illustrated in Figure 7.11, every work-item with local identifier 0 analyzes the first neighbor of u, every work-item with local identifier 1 analyzes the second neighbor of u, and so on. Therefore, this stage actually involves $|SM| \cdot MaxSMSize$ work-items, eventhough only the first κ work-items of each SM will perform a task.

Then, if there is a possible reduction of $d(v_a)$ for a neighbor v_a, every work-item that analyzed that vertex will update the positions $d(v_a)$ and $\pi(v_a)$ in the global memory. Note that this may result in |SM| simultaneous writing operations in the same memory position, but they are not critical as the values to be written are the same.

The reason for this behavior will be clarified with the description of the second stage of the algorithm but, in short, it keeps work-items in every SM active with useful information.

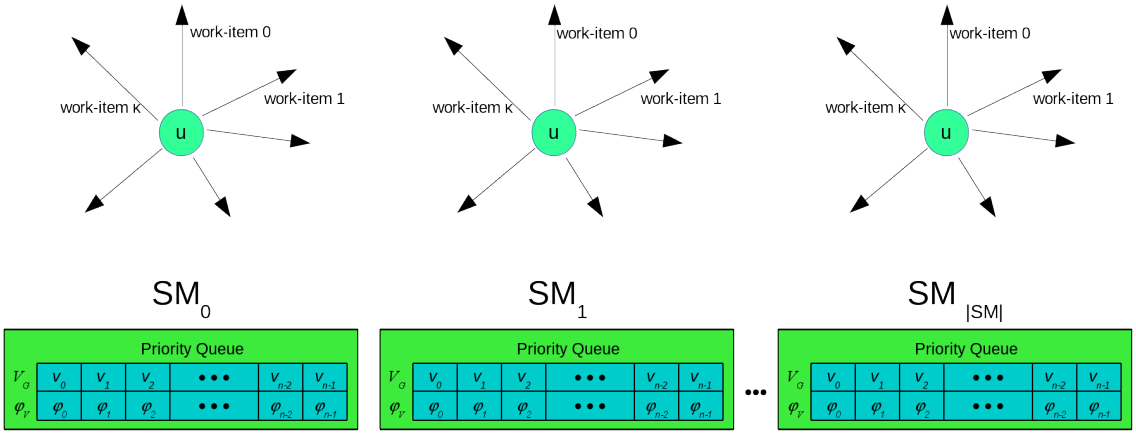


Figure 7.11: *All SMs analyzing the same vertex u.*

7.3.3 Second Stage

The next task to be performed is writing the information of all relaxed vertices in the corresponding data structures (π, d, V_G and φ). The first two, located in global memory, are updated only by one work-item.

Later, we need to write in the two data structures related to Q in the local memory: V_G and φ. At first glance, the adoption of a simplistic strategy seems to solve the problem: the chunk of Q of the first SM is populated until it is full. Next, the chunk of Q of the second SM is populated until it becomes full, then move the populating process to the third SM and so on.

However, this strategy has one main issue that can potentially increase the computation time: at each step of the algorithm, it is necessary to check the number of elements in the priority queue and whether this number has reached its maximum value. Since the inclusion of these elements occurs in a parallel fashion during the neighbor relaxation step (the first stage), the only two ways to find this amount is through a parallel reduction or using an atomically incremented variable.

To avoid the adoption of these two procedures, that would lead to an undesirable overhead in the writing operation, each *work-item*, when relaxing a vertex, only writes the information about the relaxed neighbor if its own local identifier is equal to the identifier of the SM to which the work-item belongs. This implies that the chunk of Q in every SM will receive at most one new neighbor.

This strategy avoids not only the use of the aforementioned procedures, but also splits the elements in Q through several SMs, allowing the maintainance of a variable in the local memory that will be updated in order to store the number of elements in each SM[15]. Figure 7.12 illustrates the process.

[15] An immediate effect of this strategy is that it can only be applied to graph where the value of Δ (the biggest graph outdegree) is smaller or equal the number of *SMs* available on GPU.

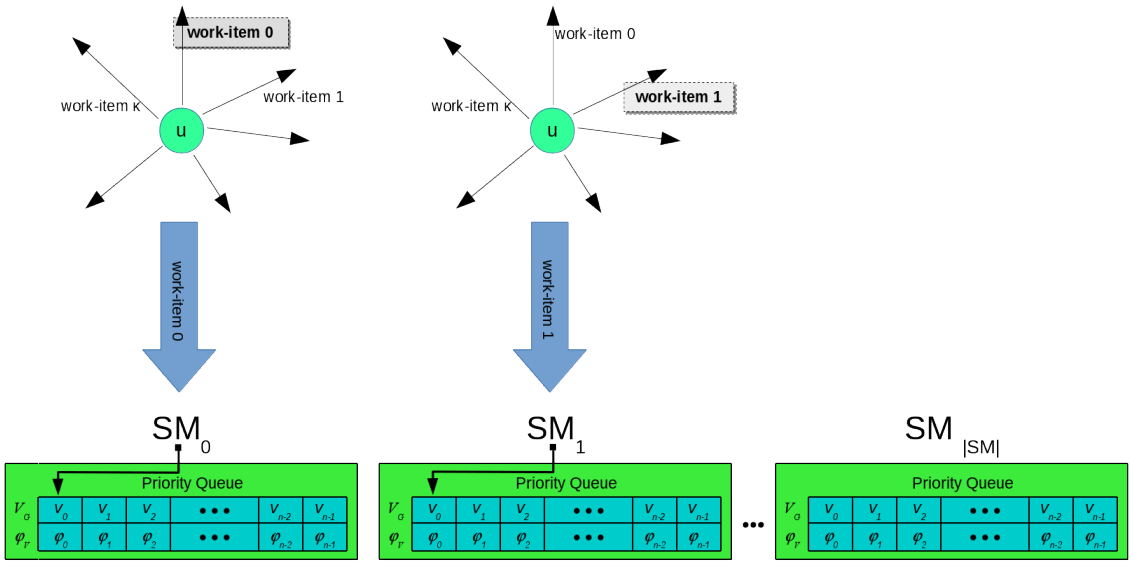


Figure 7.12: *Parallel writing in the priority queue.*

A problem arrises when adopting this strategy: if the number of SMs is significantly higher than the biggest graph outdegree ($|SM| \gg \Delta$), the chunks of Q associated with these SMs will never be used, wasting both storage space and processing power.

Solving this problem is relatively simple. Instead of operating only in the SMs with the lowest index, at each iteration of the algorithm, a circular incremental SM index is updated and used to shift the sequence of work-items that will write information. This causes the writing process to "move" between blocks of Q, distributing even more the chunk populating processing. Figures 7.13 – 7.16 depicts this strategy, assuming that 12 SMs are available in GPU and that the graph max outdegree (Δ) is equal to 3.

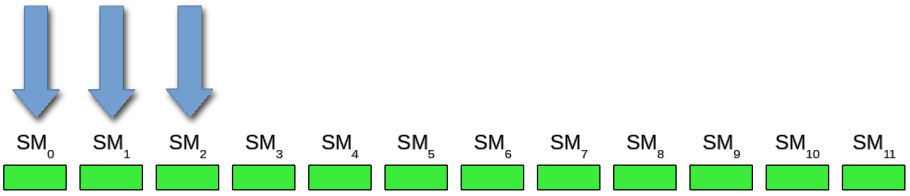


Figure 7.13: *Writing in chunks of Q – First block of active SMs.*

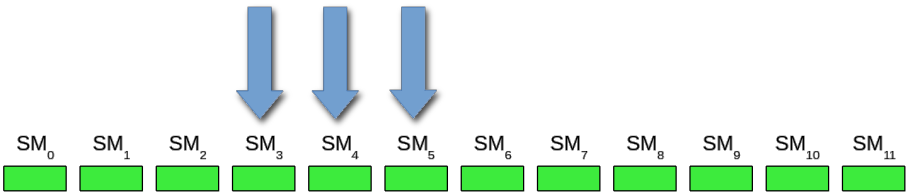


Figure 7.14: *Writing in chunks of Q – Second block of active SMs.*

This expanded distribution process leads to another, less evident, benefit: the number of elements in each chunk will grow even more slowly, distributing in a more

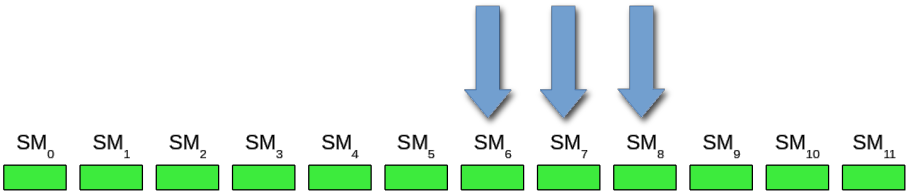


Figure 7.15: *Writing in chunks of Q – Third block of active SMs.*

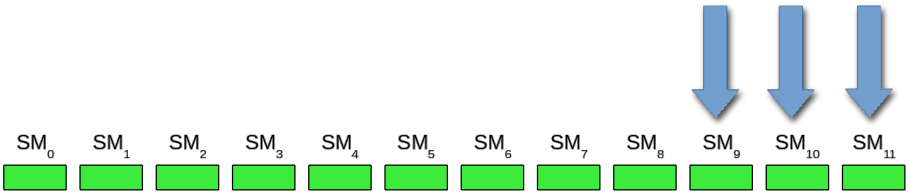


Figure 7.16: *Writing in chunks of Q – Fourth block of active SMs.*

uniform fashion the workload that the work-items should perform in the third stage of the parallel reduction, as described next.

7.3.4 Third Stage

In this last stage, the next vertex u with smallest tentative distance must be found (Line 18 of Algorithm 7.3), removed from the corresponding chunk of Q (Line 20) and used as a new frontier. To do this, the two-stage parallel reduction strategy described in Chapter 6 is employed, but modified in order to locate the smallest element with its respective vertex identification, and operating only in the elements already in local memory. Hence, lines 9 – 16 of Listing 6.8 are not necessary here, significantly decreasing the execution time of this procedure.

It is worth a short comment on how the global synchronization barrier between the two stages of parallel reduction (Line 18) is implemented in this stage. Unlike the strategy presented in Chapter 6, where a global synchronization is performed by CPU, here the work-items across multiprocessors are kept synchronized through the use of the mechanism called "*GPU Lock-Free Synchronization*" described in [253].

As a final note it is important to note that, during the implementation of these three stages of parallel Dijkstra, only one kernel was employed. The strategy is described in Algorithm 7.3.

Algorithm 7.3: $ParallelDijkstra(G,s)$

Input: Compact representation of directed weighted graph $G=(V,E)$, a vector W of edge weights and a starting node s.

Output: Sets of distances $d(v)$ and predecessors $\pi(v)$.

1 Define the current node $u \leftarrow s$, the tentative distance $\varphi(v)$, the vertex identifier V_G, the *work-item* local identifier L_{id} and the *SM* unique identifier SM_{id};
2 $initialization()$;
3 $\zeta \leftarrow 0$;
4 $\xi \leftarrow \frac{|SM|}{\Delta}$;
5 **for** *each thread* j, $j = 0,\ldots,|SM| \cdot MaxSMSize - 1$ **do in parallel**
6 **while** $|Q| \neq 0$ **do**
  `// First stage.`
7 $k_1 \leftarrow neighborsLowerBound(u)$;
8 $k_2 \leftarrow neighborsUpperBound(u)$;
9 $\kappa \leftarrow k_2 - k_1 + 1$;
10 **if** $L_{id} < \kappa$ **then**
11 **if** $d(u) + W_E < d_{v_a}$ **then**
  `// Second stage.`
12 **if** $(\frac{SM_{id}}{\xi} = \zeta)$ ***and*** $(L_{id} + \xi \cdot \zeta = SM_{id})$ **then**
13 $\pi(v_a) \leftarrow u$;
14 $d(v_a) \leftarrow d(u) + W_E$;
  `// Adds the vertex to chunk of Q.`
15 $V_G \leftarrow V_G \cup v_a$;
16 $\varphi \leftarrow \varphi \cup (d(u) + W_E)$;
17 $localSyncBarrier$;
  `// Third stage.`
18 $u \leftarrow parallelReduction2LocateSmallestElement()$;
19 $globalSyncBarrier$;
20 $removeSmallestElement(u)$;
21 $\zeta \leftarrow \zeta + (-\zeta \cdot (\zeta = (\xi - 1))) + (\zeta < (\xi - 1))$;

The initialization process in Line 2 of Algorithm 7.3 makes use of the "Persistent Threads" and "Loop Unrolling" strategies for populating all vectors with their initial values (Lines 1 – 4 of Algorithm 7.1). It also sets two private variables for each work-item. The first one, ζ, which initial value is 0, indicates the current block of active SMs. The second one, ξ, is the total number of SMs blocks, defined as $\frac{|SM|}{\Delta}$. After each iteration of the "while" loop in Line 6, the value of ζ is updated by the algebraic expression at

Line 21. This is done in this way to avoid the use of conditional instructions.

Lines 7 – 10 ensure that only the first κ work-items of each SM will relax the κ neighbors of u. The cheking at Line 11 verifies whether the neighbor of u under analysis by the work-item can be improved. If so, Lines 13 – 14 update the shortest path information accordingly.

The test in Line 12 assures that only the work-items of the active block of SMs will insert the relaxed node information in the respective chunk of Q.

Obviously, the present method only works when the maximum outdegree (Δ) of the graph is smaller or equal to $|SM|$ and to $MaxSMSize$, what is true for most road networks and GPU hardware.

7.3.5 Complexity Analysis

As can be observed, the algorithm has an initialization step, which can be carried out in parallel in time $\frac{n}{MaxSMSize \cdot |SM|}$, where $n = |V|$. Since $MaxSMSize$ and $|SM|$ are constants, this implies that the complexity of this step can be expressed by $O(n)$.

The main iteration is performed by loop in Line 6, which involves the location of the smallest element u in Q, its extracting and the relaxation of its neighbors until all chunks of Q become empty.

As each work-item operates on one neighbor of u, the relaxation process (Lines 7 – 11 of Algorithm 7.3) of all neighbors of u can be performed in constant time $O(1)$.

Something similar happens when each work-item updates the information about the relaxed neighbor $\pi(v_a)$ and $d(v_a)$ and then inserts this information in Q (Lines 12 – 16). Considering that all work-items are operating in parallel and have exclusive access to the associated chunk of Q, this operation can also be performed in constant time $O(1)$.

The next task to be carried out is finding the smallest element. As previously stated, this operation is performed only in the elements already in local memory. Hence, the three steps required to find the smallest element can be described as follows:

1. Each work-item find its local minimum in the associated chunk of Q. This operation takes time $\frac{|PQ_{chunk}|}{MaxSMSize}$;
2. Next, |MaxSMSize| smallest elements are reduced to one simple value in time $\log(MaxSMSize)$;
3. Finally, |SM| smallest elements are again reduced in order to find the global minimum. This can be done in time $log(|SM|)$.

The last task to be performed is the removal of smallest element u from the chunk of Q in which it is located. This can be done by simply copying the values stored in the last position over the position where u is located and then decreasing the local variable

that holds the number of elements in this specific chunk of Q, as illustrated in Figure 7.17. As can be easily observed, this operation is carried out in constant time – $O(1)$.

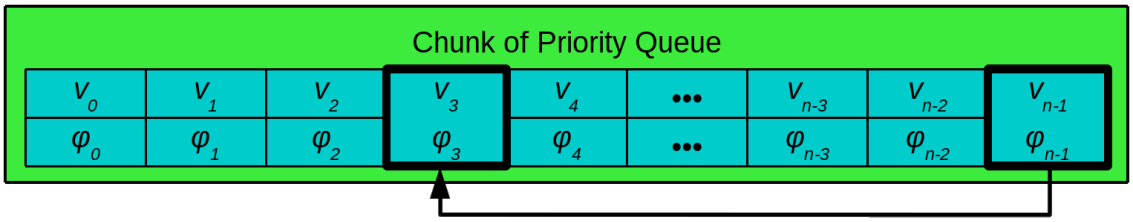


Figure 7.17: *Removing the smallest element from chunk of Q.*

So, the time required by this parallel version of Dijkstra's algorithm (including the initialization process, which is performed in n iterations) is upper bounded by two main steps:

1. The iteration of the "while" loop in Line 6, which requires a number of steps proportional to the number of elements in Q, which is, in the worst case, n.
2. The time to locate the smallest element.

Therefore, the whole algorithm requires time that is upper bounded by the expression $n \cdot \left(\frac{|PQ_{chunk}|}{|SP|} + \log(MaxSMSize) + log(|SM|) \right)$. Table 7.6 summarizes the information about all these steps.

Operation	Complexity
Initialization	$O(n)$
Relax neighbor	$O(1)$
Insert in priority queue	$O(1)$
Find the smallest element	$O\left(\frac{\|PQ_{chunk}\|}{MaxSMSize} + \log(MaxSMSize) + log(\|SM\|) \right)$
Remove from priority queue	$O(1)$
Iterate over priority queue	$O(n)$
Total	$O\left(n \cdot \left(\frac{\|PQ_{chunk}\|}{\|SP\|} + \log(MaxSMSize) + log(\|SM\|) \right) \right)$

Table 7.6: *Parallel Dijkstra: complexity analysis.*

7.4 Computational Experiments

In order to evaluate the efficiency of the proposed strategy in terms of processing time to find the shortest paths, a sequential Dijkstra algorithm was implemented in C++ for comparison. The sequential algorithm uses a binary heap because of its ease of implementation and low overhead in execution time [51].

As pointed out by [43], although Fibonacci heaps have a smaller asymptotic complexity compared to binary heaps, in real applications often binary heaps lead to algorithms with lower execution times, due to Fibonacci's large constant factors [60].

The machine used in the tests and the compiler parameters were the same as those described in Section 5.4.

The experiments were performed on 15 graphs representing large USA road networks meshes, obtained from the "*Stanford Large Network Dataset Collection (SNAP)*" [160] and from the "*9th DIMACS implementation challenge – Shortest Paths*" [69].

For each graph, a set of 100 nodes were randomly chosen and then fixed as starting points. For each node, a sequence of 5 executions were performed and the average time of these 500 executions (100 nodes · 5 executions) was computed. Table 7.7 shows information about the graphs and the computational results. The first columns are the identification of the graphs and their number of vertices and edges. The next columns are the average time (in seconds) of the sequential Dijkstra algorithm (T_{seq}), the average time (in seconds) of the parallel algorithm (T_{par}) and the speedup ($\frac{T_{seq}}{T_{par}}$).

	Graph name	$\|V\|$	$\|E\|$	$T_{seq}(s)$	$T_{par}(s)$	Speedup
1	New York City	264.346	733.846	0,085	0,00941	9,032
2	San Francisco Bay Area	321.270	800.172	0,099	0,01063	9,317
3	Colorado	435.666	1.057.066	0,139	0,01179	11,786
4	Florida	1.070.376	2.712.798	0,365	0,02110	17,297
5	Pennsylvania	1.088.092	3.083.796	0,437	0,02119	20,627
6	Northwest USA	1.207.945	2.840.208	0,423	0,01949	21,709
7	Texas	1.379.917	3.843.320	0,549	0,02392	22,949
8	Northeast USA	1.524.453	3.897.636	0,561	0,02473	22,684
9	California and Nevada	1.890.815	4.657.742	0,736	0,02537	29,006
10	California	1.965.206	5.533.216	0,904	0,02836	31,875
11	Great Lakes	2.758.119	6.885.658	1,076	0,02944	36,546
12	Eastern USA	3.598.623	8.778.114	1,399	0,03520	39,739
13	Western USA	6.262.104	15.248.146	2,538	0,06265	40,508
14	Central USA	14.081.816	34.292.496	7,329	0,17973	40,779
15	Full USA	23.947.347	58.333.344	10,619	0,25990	40,859

Table 7.7: *Dijkstra: Sequential and Parallel Execution Times*

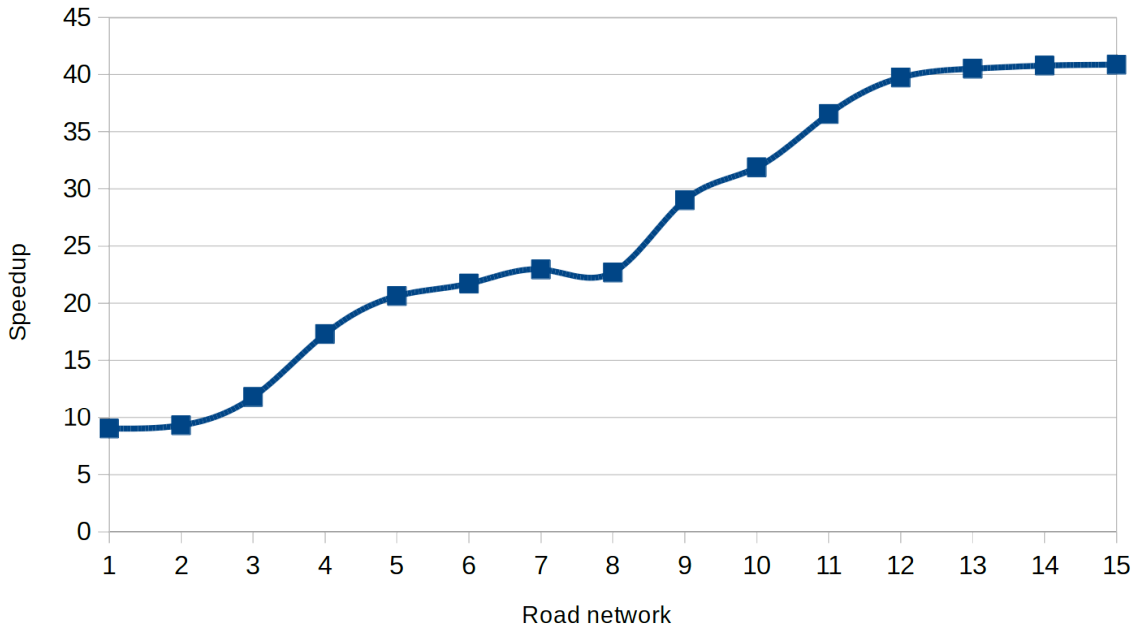


Figure 7.18: *Parallel speedup according to road network.*

It is worth remembering that the parallel times shown in Table 7.7 do not consider the time for copying the generated data back to the CPU. This was done because during the urban traffic allocation process described in Chapter 8, these data are only used as support at an intermediate stage in the task of demand allocation. If this copy was needed, the speedups would be smaller. For this reason, the impact of the copy process was not taken into account by the present study.

7.4.1 Analysis of the Results

From the results shown in Table 7.7, it can be seen that, as the size of the road network increases (in number of nodes and arcs), greater benefit is achieved through the use of GPU parallelism with the described technique. For example, the first line of Table 7.7 shows that, for the city of New York's road mesh the advantage of the parallel algorithm in relation to the sequential one was approximately 9x with respect to its execution times. On the other hand, in the graph representing the entire North American road mesh (Full USA) – a lot bigger than the first one – the obtained speedup was close to 41x.

This clearly shows that GPU scalability is better than CPU scalability, that is, GPU performance is less affected as the workload assigned to it increases, as stated by Gustafson's Law (see Section 3.1.2). Chart 7.18 illustrates the speedup improvement achieved by the proposed implementation over the sequential version.

It is worh mentioning that, for the traffic assignment process discussed in Chapter 8, the results of the parallel Dijkstra algorithm are immediately used by another

parallel stage inside the GPU, without having to be transfered to the CPU RAM (thus, the communication time, T_{com}, is null). Therefore, the mentioned speedups are real for our application.

7.5 General Remarks

This chapter presented the problem of computation of shortest paths in weighted graphs. The four main variants (P2P, single source, many to many and all pairs) of this problem were described, being evaluated according to their applicability to real problems. Advantages and disadvantages of each one were highlighted, demonstrating that for the present work, the SSSP approach is the most adequate one, due to the lower memory usage.

Among the two possible algorithms to solve the SSSP, the Dijkstra approach was selected because of its lower computational time complexity when compared to that of Bellman-Ford-Moore, taking into account that the costs of the arcs are always positive in road networks, since they usually represent travel times.

Several parallel approaches to the Dijkstra and Bellman-Ford-Moore algorithms were evaluated, concluding that none of them were satisfactorily adapted to the problem under study by the present work. As a result, a thorough analysis of the behavior dynamics of the Dijkstra priority queue, when applied to representative graphs of road networks, was conducted. At the end, it was verified that this behavioral dynamics allows the conception of a new parallel approach for typical road networks. The same strategy can be used for other sparse graphs with similar structure (close to a grid and with large diameters).

The performed experiments demonstrated a clear advantage of this new approach when compared to the sequential version, by efficiently exploiting the GPU resources, such as memory hierarchy, to deal with the main bottleneck in Dijkstra's parallelization: its priority queue.

As will be shown in the next chapter, the computation of shortest paths is one of the most costly tasks with respect to the runtime required in the macroscopic traffic allocation process. This is due to the need of solving the SSSP for every source vertex of the network and for each iteration of the traffic assignment algorithm. Therefore, a fast parallel Dijkstra algorithm is of much value to the simulation and the study of urban traffic conditions.

CHAPTER 8

GPU Computing Applied to the Traffic Assignment Problem

This chapter presents how a macroscopic traffic assignment can be achieved through the use of Beckman's model and discusses a GPU-based implementation of it. It also shows comparative experiments between its sequential and GPU parallel versions when applied to several large graphs representing road networks. The chapter is organized as follows: Section 8.1 gives some details about Beckman et al.'s model, describes the arc cost functions used in the present work and depicts some methods for determining the equilibrium point in transportation networks. Section 8.2 presents a profiling analysis of a sequential algorithm and highlights its most time consuming steps. Section 8.3 explains how the sequential algorithm was adapted in order to explore the GPU capabilities. Section 8.4 shows the experiments performed and, finally, Section 8.5 draws the conclusions.

8.1 Background

Here we extend the basic definitions given in Section 2.2. The mathematical modeling of Beckmann et al. [11] considers the Traffic Equilibrium Problem in the condition of System Optimization (TEP-SO) as a minimization problem, defined as follows:

$$\begin{aligned} min_x \quad & \sum_{a \in E} x_a \cdot t_a \\ subject\ to \quad & x = \Lambda \cdot f \\ & d = D \cdot f \\ & f \geq 0 \end{aligned}$$

where

- x_a represents the flow in arc a, grouped in vector x;
- t_a is the function that describes the time needed to travel arc a, based on its physical characteristics and current flow. t_a must be a convex, continuous, non-negative and non-decreasing function;
- f is a vector that contains the flows in all paths between origin and destination nodes;
- d is the vector, presumably known, of demands between source and destination nodes.
- Λ and D associate arcs, paths and demand for an O-D (origin-destination) pair. A flow f in paths or x in arcs is considered feasible if the demand d is met;
- The term $\sum_{a \in E} x_a \cdot t_a$ is the total flow time in seconds.

As explained in Section 2.2.2, the TEP-SO aims at finding the ideal flow that minimizes the total travel time and also fuel consumption, eventhough it may result in a longer travel time for some individual drivers.

Being $T_k = \sum_{a \in k} t_a$ the sum of the travel times in the arcs that form a path ρ, that is, the total travel time from a certain source to a destination, the first Wardrop Principle can be expressed as $T_k(f^*)(f - f^*) \geq 0$ for all feasible flows f of the O-D pair k. It can be shown [11, 197] that this corresponds to the optimality conditions of a convex optimization problem, previously defined as TEP-UE (User Equilibrium) [47]:

$$\begin{aligned} min_x \quad & \sum_{a \in E} \int_0^{x_a} t_a \, dx \\ subject\ to \quad & x = \Lambda \cdot f \\ & d = D \cdot f \\ & f \geq 0 \end{aligned}$$

8.1.1 The Arc Types and its t_a Functions

The central point in Beckman et al. model (Section 2.2.2) is the construction of one or more functions that describe the average time taken by the vehicles to travel through a certain portion of road network (t_a functions). However, the original model of Beckmann assumes that the functions should only consider the physical characteristics and traffic flows in the arc itself (i.e., a *separable* function), ignoring effects exerted by vehicle flows in other arcs (i.e., a *non-separable* function).

An arc cost function τ is said to be *separable* when the cost of arc a, expressed by τ_a, only depends on the existing traffic flow (x_a) in this arc. Hence, $\tau_a(x_a)$ can be computed using just the x_a value[1], that is, flow that travels on the arc a in the considered time interval.

On the other hand, τ_a is said as *non-separable* when it depends on a set of existing traffic flows on several arcs that form the road network, not just the traffic flow in a itself. Then $\tau_a(\rho) = \tau_a(f_0, f_1, f_2, ..., f_n)$, where ρ is a vector composed of all traffic streams that are used to compute τ_a in a given time interval.

Although the original approach of Beckman et al. can be suitable for arcs that do not interact with other flows, for example free ways, it ends up not being suitable for those streets/avenues that are actually influenced by the movement of other vehicles. Among these, non-preferred avenues may be cited, where drivers must wait their turn before continuing the journey, or those that end on a roundabout.

Therefore, for a better approximation to reality, the t_a functions should consider such influences during the travel time calculation. Next the description of the arc types and their associated t_a function are presented. These functions are improvements over the ones initially proposed in [64] and later refined in [141, 142]:

- **Type 1** (one-way, ending with a traffic light) and **Type 2** (two-way, no interaction, ending with a traffic light):

$$t^{1,2} = \frac{3.6 * c_a}{v_a} \left[1 + \frac{x_a}{2500 * f_a} + \left(\frac{x_a}{500 * f_a} \right)^3 \right] + s * \left(3 + \frac{3.6 * c_a}{v_a} \right)$$

- **Type 3** (two-way, with interaction, ending with a traffic light):

$$t^3 = \frac{3.6 * c_a}{v_a} \left[1 + \frac{3x_a + z}{8000 * f_a} + \left(\frac{x_a}{900 * f_a} \right)^3 + \left(\frac{z}{1500 * f_a} \right)^3 \right] + s * \left(3 + \frac{3.6 * c_a}{v_a} \right)$$

- **Type 4** (one-way, preferential) and **Type 5** (divided and preferential):

$$t^{4,5} = \frac{3.6 * c_a}{v_a} \left[1 + \frac{x_a}{4000 * f_a} + \left(\frac{x_a}{900 * f_a} \right)^3 \right]$$

- **Type 6** (two-way, with interaction with opposite hand):

$$t^6 = \frac{3.6 * c_a}{v_a} \left[1 + \frac{3x_a + z}{14000 * f_a} + \left(\frac{x_a}{900 * f_a} \right)^3 + \left(\frac{z}{1600 * f_a} \right)^3 \right]$$

[1] An the physical characteristics of the arc a.

- **Type 7** (one-way, non-preferential) and **Type 8** (divided, non-preferential. y_1 and y_2 are preferential flows):

$$t^{7,8} = \frac{3.6 * c_a}{v_a}\left[1 + \frac{x_a}{2500 * f_a} + \left(\frac{x_a}{900 * f_a}\right)^3\right] + \left(\frac{y_1}{300}\right)^2 + \left(\frac{y_2}{300}\right)^2$$

- **Type 9** (two-way with interaction, non-preferential. y_1 and y_2 are preferential flows):

$$t^9 = \frac{3.6 * c_a}{v_a}\left[1 + \frac{3x_a + z}{10000 * f_a} + \left(\frac{x_a}{900 * f_a}\right)^3 + \left(\frac{z}{1600 * f_a}\right)^3\right] + \left(\frac{y_1}{300}\right)^2 + \left(\frac{y_2}{300}\right)^2$$

- **Type 10** (one-way, small loop at final node) and **Type 11** (two-way, no interaction, small loop at final node):

$$t^{10,11} = \frac{3.6 * c_a}{v_a}\left[1 + \frac{x_a}{3000 * f_a} + \left(\frac{x_a}{1200 * f_a}\right)^3 + \left(\frac{w_1}{1800}\right)^3 + \left(\frac{w_2}{1800}\right)^3 + \left(\frac{w_3}{1800}\right)^3\right]$$

- **Type 12** (two-way, with interaction, roundabout at final node):

$$t^{12} = \frac{3.6 * c_a}{v_a}\left[1 + \frac{3x_a + z}{12000 * f_a} + \left(\frac{x_a}{1200 * f_a}\right)^3 + \left(\frac{w_1}{1800}\right)^3 + \left(\frac{w_2}{1800}\right)^3 + \left(\frac{w_3}{1800}\right)^3 + \left(\frac{z}{2000 * f_a}\right)^3\right]$$

where

- c_a is the length (in meters) of the arc a, representing a portion of the complete road/street/avenue;
- v_a is the free speed (i.e., maximum allowed) in km/h of a;
- f_a is the number of lanes of a;
- x_a is the traffic flow (in one hour) in the arc a;
- z is the traffic flow in the reverse arc, in roads/streets without division between lanes (in one hour);
- y_1 and y_2 are the flows in preferential arcs, intersecting the end of arc a, in one hour. See Figure 8.1 for an illustration;
- w_1, w_2 and w_3 are clockwise flows intersecting the end of arc of types 10, 11 or 12 (in one hour). See Figure 8.2;

- s_a is the traffic light factor. This is a value that represents the delay for traveling the current arc, and must be estimated to make $s_a * \left(3 + \frac{3.6 * c_a}{v_a}\right)$ the average delay time that a traffic light imposes to the vehicle flows, based on field measurements.

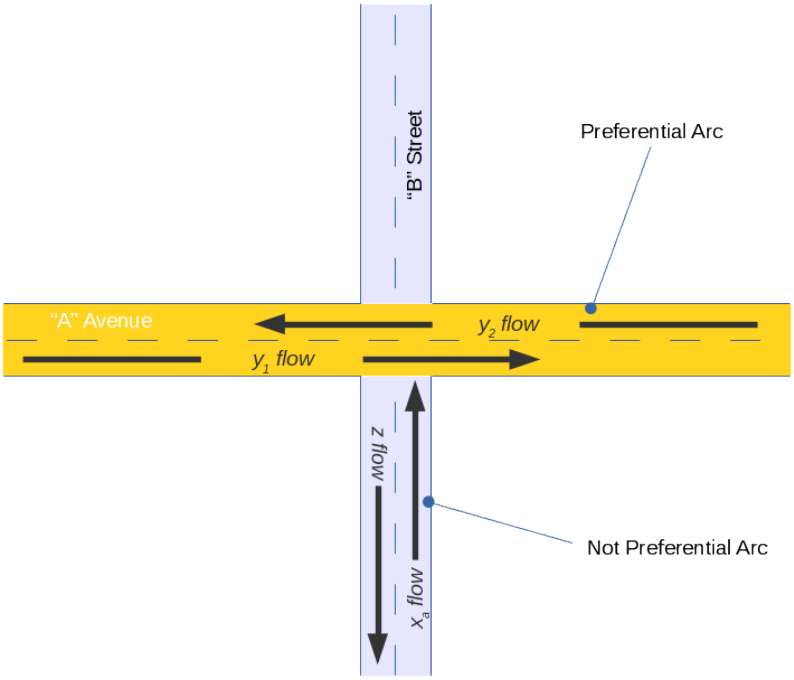


Figure 8.1: *Arc cost function: considering flows in preferred ways.*

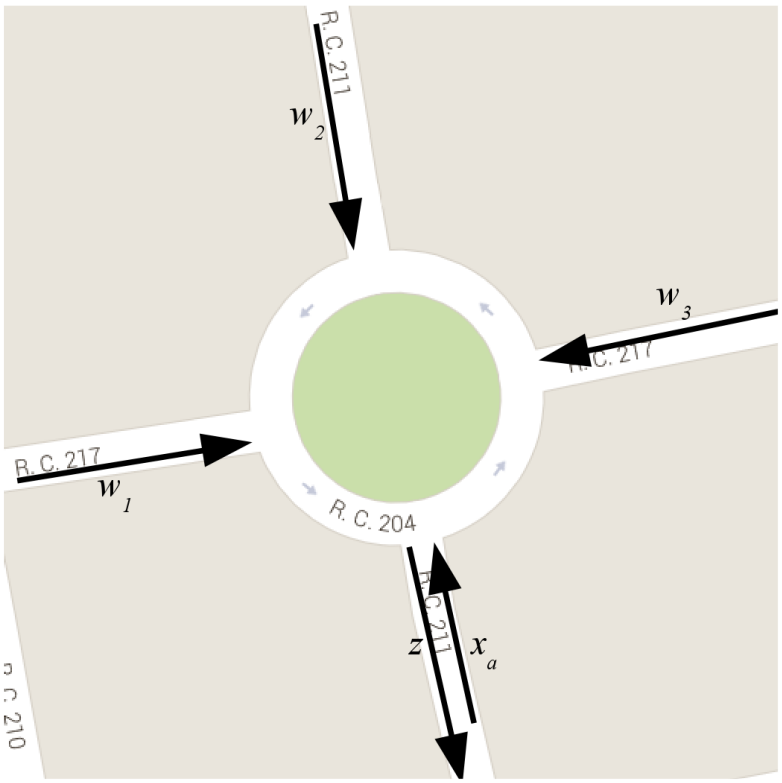


Figure 8.2: *Arc cost function: roundabout flows.*

8.1.2 Methods for Determining the Equilibrium Point in Transportation Networks

There are several algorithms that solve the aforementioned problems following distinct mathematical approaches. Among them, "Gradient Projection" [211, 15] and "Convex Combination" (also known as Frank-Wolfe (FW) algorithm) [96, 31, 181] methods can be cited. These two methods are usually labeled as "feasible directions methods" since their search strategy always points in the direction where some potencial (viable) solution can be found. However, they use very different ideas to model the network equilibrium and a detailed comparison between them can be found in [52].

As pointed out by Costodio [52], the natural choice to solve the equilibrium problem in a transportation network falls on a convex combination method due to its low memory consumption. Thanks to its efficiency, Sheffi [225, 7], in his book, proposes the use of this method in an algorithmic solution to model the equilibrium on transportation networks. Such algorithm, used in our work, involves several steps presented next in detail and further information about it can also be found in the literature [225, 7, 52]:

- **Step 1**: *Initialization.* $i \leftarrow 0$. Find a feasible x^0 initial flow;
  In a first moment, it is possible to assume that there are no vehicles moving around the road network, only a certain amount of cars that want to travel from a node **A** to a node **B**, ie a *demand of flow*.
  In this case, with the initial flows equal to zero, all t_a functions are reduced to the first term of their equations,

$$\frac{3.6 \cdot c_a}{v_a}$$

  which is the necessary time (in seconds) to travel arc a in an condition of completely free road. Using these times as the arc costs (or weights), the shortest paths between all $O-D$ pairs are computed. This can be done using one of the algorithms mentioned in Chapter 7.
  Once computed the shortest paths, an *all-or-nothing* demand assignment is performed. In other words, all vehicles that wish to move from one point to another will follow the shortest path between these two points. Figure 8.3 illustrates the all-or-nothing assignment for a simple graph with three demands (the corresponding O-D matrix is shown in Table 8.1). The result of this process is the initial feasible flow x^0.
- **Step 2**: *Update.* Calculate $t_a^i(x_a^i)$, for $\forall\, a \in E$;
  With the viable flow $x^i(i=0)$ in hand, the travel times are estimated again. As now the vehicle flows are non-zero, their travel times will be different from the first calculation, and other shortest paths are computed for the demands.

- **Step 3**: *Direction find.*
  With the t_a^i times as arc costs, find the shortest paths between all O-D pairs. Next, apply the same all-or-nothing assignment employed at Step 1 to compute a new set of arc flows, called here y^i (with y_a^i for each arc a);
- **Step 4**: *Line search.* Find $\alpha^i \in \mathbb{R}$ in [0,1] interval that minimizes the function:

$$\sum_{a=1}^{|E|} \int_0^{x_a^i+\alpha(y_a^i-x_a^i)} t_a^i(x_1^i, x_2^i, \ldots, \omega, \ldots, x_m^i) d\omega \quad \text{for User Equilibrium or}$$

$$\sum_{a=1}^{|E|} x_a^i * t_a^i(x_1^i, x_2^i, \ldots, \omega, \ldots, x_m^i) \quad \text{for System Optimization}$$

  Methods for this step were described in Section 8.1.2.
- **Step 5**: *Move.* Set $x^{i+1} \leftarrow x^i + \alpha^i(y^i - x^i)$;
- **Step 6**: *Convergence test.* Check the convergence ε;
  If $|x^{i+1} - x^i| < \varepsilon$, the algorithm finishes. Otherwise, $i \leftarrow i+1$ and back to Step 2.

In this simple example, there are three vehicles demands. Their values and shortest paths are:

A ⇒ E (A → B → C → E) : 100 vehicles
D ⇒ E (D → B → C → E) : 50 vehicles
A ⇒ B (A → D → B) : 75 vehicles

In the first flow assignment, the A ⇒ E demand of 100 vehicles travels the shortest path toward its destination vertex;

In the second assignment, the D ⇒ E demand of 50 vehicles is distributed along its shortest path. Since arcs B → C and C → E already have cars traveling through them, their new flow values are now 150;

Finally, the third assignment demand A ⇒ B is performed. The arc D → B already has 50 cars, and its flow becomes 125.

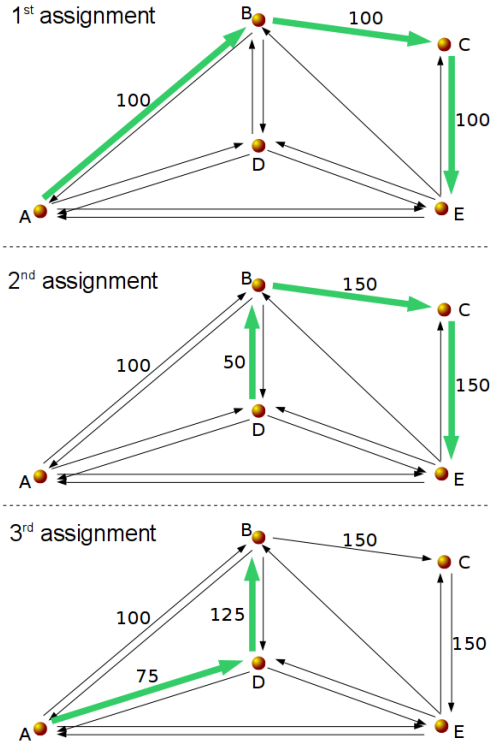


Figure 8.3: *Macroscopic traffic allocation: flow assignment through the shortest paths.*

	A	B	C	D	E
A	0	75	0	0	100
B	0	0	0	0	0
C	0	0	0	0	0
D	0	0	0	0	50
E	0	0	0	0	0

Table 8.1: *Example of an O-D matrix with 3 demands.*

Methods for Step 4

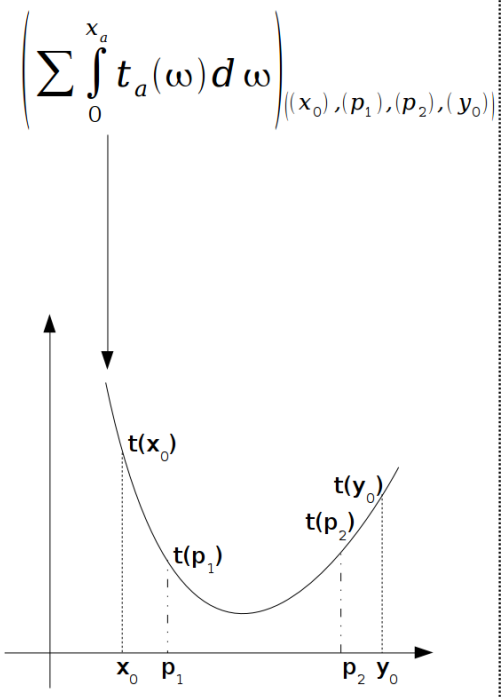
As described in Section 2.2.3 and at the begining of Section 8.1, the mathematical modeling of Beckmann et al. considers TEP-UE and TEP-SO as minimization problems, which objective functions are nonlinear but convex.

Since the present work makes use of a "Convex Combination" method, the current section depicts three techniques for finding the extremum (minimum or maximum) of *unimodal* functions, where the objective functions of TEP-UE and TEP-SO are categorized. In all of them, the sustaining ideia is to, starting with a known interval, successively narrow the range of values inside that interval, where the extremum is known to exist.

Formally, a convex function *f(x)* is defined as *unimodal* if, given some m, it is monotonically decreasing for $x \leq m$ and monotonically increasing for $x \geq m$. Therefore, the minimum value of $f(x)$ can be found at $f(m)$ and there are no other local minimums [70]. A concave function can be defined in a similar way.

The possible three techniques that can be used to minimize the objective function of the TEP-SO/TEP-UE, in Step 4 of the general algorithm, are:

- **Golden Ratio** – Developed by Kiefer [149], the basic idea of this method is to determine an interval [a, b] containing the minimum of a function, so that it satisfies the relation $b - a < \varepsilon$, where $\varepsilon > 0$ is small enough and $\theta = \frac{1+\sqrt{5}}{2} \approx 1.618033988$ is the *golden ratio*, as shown in Figure 8.4.
- **Fibonacci** – It is a search method also proposed by Kiefer [149], similar to the golden ratio. The author used the numbers of the Fibonacci sequence to develop a convex function optimization method and named it as Fibonacci Search [189]. The sequence of Fibonacci numbers is given by the serie $F_n = F_{n-1} + F_{n-2}$, where $F_0 = F_1 = 1$. Taking I_k as the range of uncertainty after k iterations and I_0 the initial range, we have: $I_k = \left(\frac{F_{N-k}}{F_N}\right) \cdot I_0$ where F_N is the Fibonacci number (N is an integer number initially set), and $k = 1, 2, ..., N - 1$. The discovery process of the points that determine the uncertainty interval is illustrated in Figure 8.5.
- **Dichotomy** – This optimization method is characterized by the reduction by half of the search range at each iteration. Assuming that the desired point belongs to the



Initially, an interval [x0, y0] is determined. Then, two points defined by

$$p_1 = \theta x_0 + (1-\theta) * y_0$$
$$p_2 = (1-\theta) * x_0 + \theta y_0$$

are inserted within this range and the function values at points x_0, y_0, p_1 and p_2 are computed.

- If $t(x_0) < t(y_0)$ then y_0 is discarded, p_2 becomes the new y_0 and another $p_2 = (1-\theta) * y_0 + \theta x_0$ is computed;
- If $t(x_0) > t(y_0)$ then x_0 is discarded, p_1 becomes the new x_0 and another $p_1 = (1-\theta) * x_0 + \theta y_0$ is computed.

The process repeats until the relation

$$y_0 - x_0 < \varepsilon$$

is satisfied.

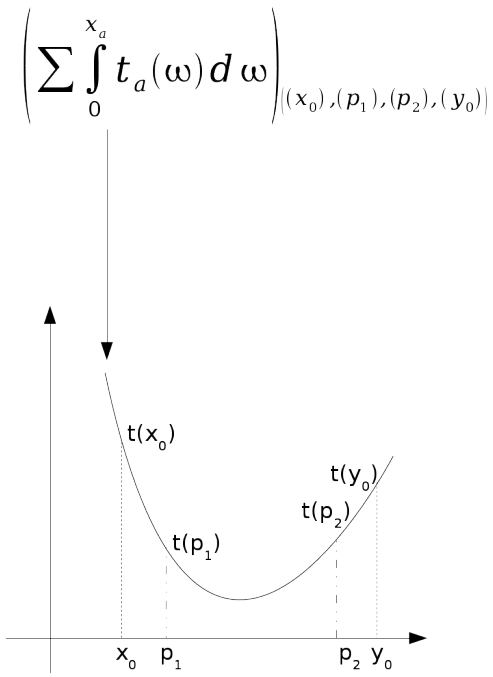
Figure 8.4: *Method of feasible directions: Golden Ratio.*

interval [a, b], the method is based on the existence of a triplet (x_1, x_3, x_5) such that $f(x_1) > f(x_3) < f(x_5)$ and that the desired point is located at $[x_1, x_5] = [a, b]$. To get x_3, the expression $x_3 = \frac{x_1+x_5}{2}$ is used. Such a point divides the interval $[x_1, x_5]$ into two sub-intervals of equal size: $[x_1, x_3]$ and $[x_3, x_5]$. Next, the points x_2 and x_4 are generated, with $x_2 = \frac{x_1+x_3}{2}$ and $x_4 = \frac{x_3+x_5}{2}$. The resultant objective values are then compared. If $f(x_1) > f(x_2) < f(x_3)$, the new triplet will be (x_1, x_2, x_3); otherwise, its value becomes (x_3, x_4, x_5). The process loops until the interval is smaller than a previously defined ε.

Novaes [189] compared these methods, considering its precisions according to the number of evaluations of the objective function, highlighting the superiority of the Fibonacci and the Golden Ratio methods in comparison to Dichotomy. He points the Golden Ratio as the best option because of its ease implementation and very close precision to the Fibonacci Search. Therefore, such method is also adopted in the current study.

8.2 Profiling Analysis

In order to identify the most problematic (time consuming) parts of the algorithm described in Section 8.1 of the program during the sequential process of macroscopic urban traffic assignment, the present study implemented it as a computer program and



1. Initially it is defined an interval [x0, y0].
2. Two points are inserted within that range, symmetrically

$$I_1 = \left(\frac{F_{N-1}}{F_N}\right) \times I_0$$

far of the extremities and the value of the function at the points p_1 and p_2 is estimated.

3. If $t(x_0) < t(y_0)$, $\boldsymbol{p_2}$ becomes the new $\boldsymbol{y_0}$;
   If $t(x_0) > t(y_0)$, $\boldsymbol{p_1}$ becomes the new $\boldsymbol{x_0}$;

Determined the new interval, return to step 2. The process repeats until the relation

$$\frac{1}{F_N} < \varepsilon$$

is satisfied, where F_N is the Fibonacci number and ε is the desired precision.

Figure 8.5: *Method of feasible directions: Fibonacci Search.*

conducted a detailed analysis of its execution through the use of a profiler. These problematic points are the ones that deserve special attention for parallelization.

The sequential program was coded in the C++ language and compiled using a GNU compiler (g++ version 4.8.2 with parameters "`-O3 -mcmodel=medium -m64 -g -W -Wall`"). To generate the necessary code for the profiler, the compiler parameter "-pg" was also used. With this parameter, during the program execution, detailed profiling information is created for the *GNU GCC Profiling Tool (gprof)*.

It is important to point out that, with the extra amount of code inserted in the compiled program for creating profiling information, this usually imposes a considerable overhead on the final binary code, greatly increasing its runtime. Nevertheless, it is expected that the overhead is equally distributed over the whole program.

The shortest path algorithm adopted in the program was the one proposed by Dijkstra, and its priority queue was implementated using a binary heap (see Section 7.1.2 for further details).

To collect the desired profiling information, the first ten graphs of road networks presented in Table 7.7 were used during the performed experiments. The last five graphs were not used due to the excessive time needed for their execution and profiling.

Since none of the road networks in the study had certain information necessary to the calculation of the time to travel through each arc, such as free speed, number of lanes, etc., this data was randomly defined and then saved in files on disk for later use.

To each road network a set of six O-D matrices $N \cdot N$ were assigned, with the sizes

of N defined as 1000, 2500, 5000, 7500, 10000 and 15000. To each row of all matrices a set of 256 randomly generated values was assigned. The matrices were also persisted in disk, being used later during the program execution.

For all combinations of road networks and O-D matrices, five consecutive executions of the macroscopic traffic assignment were performed. The machine used in all tests was the same described in Section 5.4.

After finishing the executions, the generated profiling information was collected and an average was calculated for each sub-routine of the code.

Table 8.2 shows the execution times (in seconds) for each road network, where the column "Sim." is the time of the simulation itself and the column "Total" is the sum of simulation time plus the time to collect the data for profiling.

Table 8.3 summarizes the profiling results for the most time consuming sub-routines. The presented percentage was was generated by summing the profiling values for each combination of road network and O-D matrix and then calculating its average value.

	1000		2500		5000		7500		10000		15000	
	Sim.	Total	Sim.	Total	Sim.	Total	Sim.	Total	Sim.	Total	Sim.	Total
New York City	1601	2346	3431	5954	5426	13988	7175	20503	7968	25002	9756	33752
San Francisco Bay Area	1916	2849	4132	7219	6568	16977	8676	24898	9637	30350	11818	40978
Colorado	2540	3862	5610	9793	8917	23095	11763	33842	13081	41246	16042	55697
Florida	6217	9318	13691	23945	21786	56586	28870	83141	31946	101265	39272	136712
Pennsylvania	6231	9376	13896	24334	22107	57433	29268	84504	32443	102847	39916	138936
Northwest USA	7555	11302	14496	28131	23364	68899	31405	99856	34654	126963	40674	168720
Texas	8606	12919	16598	32238	26777	78908	35955	114485	39689	145606	46602	193421
Northeast USA	8197	16330	16097	40303	26181	82339	31259	118300	34727	144589	41398	204538
California and Nevada	10133	20274	19952	49952	32473	102210	38709	146944	43101	179542	51349	253984
California	10598	21329	20933	52563	34154	107506	40684	154597	45280	188907	53989	267267

Table 8.2: *Sequential execution times for each road network*

Function	Time (%)	Goal
performDijkstra	44,87	Compute the shortest paths between all O-D pairs.
assignFlows	26,79	Assign all O-D flows through their shortest paths.
calcArcTimes	14,27	Compute the cost function of each arc a.
applyGoldenRatioMethod	8,91	Minimize the objective function, using the "Golden Ratio" method.

Table 8.3: *Most time consuming methods.*

As can be seen in the results, a set of only 4 tasks consumed almost 95% of the total program execution time. Other routines, such as initialization of variables, load of

the road mesh and internal sub-routines barely exceeded 5% of the total time.

About 75% of the computation time was spent in the tasks of computing the shortest paths and transforming O-D demands in traffic flows. The results presented in Table 8.3, therefore, justify the great effort employed in the present study in order to find efficient parallel solutions to solve these problems, which was done and presented in the previous chapters.

The next section describes how a GPU-based TAP was implementated.

8.3 A GPU-Based Traffic Assignment Implementation

The parallel implementation for GPU of the traffic allocation algorithm is, in fact, a substitution of the calls to the existing sub-routines in the sequential implementation described in this chapter by their respective parallel versions, that is, code able to be executed in GPUs. Therefore, there is no structural change of the general algorithm. Next, the implementation details of the overall code, as a host CPU program and of the parallel GPU subroutines for the main steps are presented.

Step 1: *Initialization.* $i \leftarrow 0$. Find a feasible x^0 initial flow;
Making all initial flows equal to zero, $|E|$ work-items are launched in order to compute the value of t for each arc a. Although the present work employs twelve different arc cost functions, the implemented code does not allow divergences in the execution flow thanks to the use of algebraic expressions, as described in Section 6.3.
Next, using these initial times as the arc costs (or weights), the shortest paths between all $O-D$ pairs are computed using the approach described in Section 7.3. Now an all-or-nothing demand assignment is performed, where each work-item handles one $O-D$ pair. This is the initial feasible flow x^0.

Step 2: *Update.* Using x^0, calc new t_a times using the same operations described in Step 1;

Step 3: *Direction find.* Find new shortest paths for all $O-D$ demands and a new feasible flow y^0;

Step 4: *Line search.* Minimize the desired function (UE or SO, see Section 2.2.2), using the strategy presented in Section 6.3.

Step 5: *Move.* Set $x^{i+1} \leftarrow x^i + \alpha^i(y^i - x^i)$;

Step 6: *Convergence test.*. Copy the value of $\rho = |x^{i+1} - x^i|$ back to CPU and check if $\rho < \varepsilon$, for ε a small tolerated error, then stop. Otherwise, do $i \leftarrow i+1$ and go back to Step 2.

8.4 Computational Experiments

As well as all the other parallel algorithms previously described, the parallel TAP was coded in the C++ language, compiled with the same parameters and tested using the machine described in Section 5.4.

The experiments to measure the performance of these codes used the same data sets and methodologies described in Section 8.2. Thus, any observed performance gains are due only to the parallel strategies employed.

The following results were split into individual tables because of the space occupied by each one. All execution times are presented in seconds. The column "*Parallel (execution)*" depicts the time needed by the parallel algorithm to perform the traffic assignment. The column "*Parallel (total)*" includes that time plus the time for all data transfers between CPU and GPU. The speed up is given by the division $\frac{Time(sequential)}{Parallel(total)}$.

New York City				
Demand Matrix Size	Sequential	Parallel (execution)	Parallel (total)	Speedup
1000	1601	173.8	174.18	9.19
2500	3431	352.9	353.26	9.71
5000	5426	413.1	413.47	13.12
7500	7175	515.4	515.80	13.91
10000	7968	546.4	546.77	14.57
15000	9756	610.3	610.61	15.98

Table 8.4: *Sequential and parallel execution times for the road network of New York City*

San Francisco Bay Area				
Demand Matrix Size	Sequential	Parallel (execution)	Parallel (total)	Speedup
1000	1916	183.0	183.40	10.45
2500	4132	405.9	406.25	10.17
5000	6568	404.1	404.53	16.24
7500	8676	406.4	406.75	21.33
10000	9637	408.2	408.62	23.58
15000	11818	407.3	407.66	28.99

Table 8.5: *Sequential and parallel execution times for the road network of New York City*

Colorado				
Demand Matrix Size	Sequential	Parallel (execution)	Parallel (total)	Speedup
1000	2540	258.3	258.83	9.81
2500	5610	577.4	577.88	9.71
5000	8917	570.6	571.12	15.61
7500	11763	579.9	580.44	20.27
10000	13081	581.5	582.04	22.47
15000	16042	582.6	583.11	27.51

Table 8.6: *Sequential and parallel execution times for the road network of Colorado*

Florida				
Demand Matrix Size	Sequential	Parallel (execution)	Parallel (total)	Speedup
1000	6217	335.3	335.78	18.52
2500	13691	735.5	735.99	18.60
5000	21786	833.8	834.34	26.11
7500	28870	928.5	929.03	31.08
10000	31946	1025.5	1026.06	31.13
15000	39272	1157.5	1158.06	33.91

Table 8.7: *Sequential and parallel execution times for the road network of Florida*

Pennsylvania				
Demand Matrix Size	Sequential	Parallel (execution)	Parallel (total)	Speedup
1000	6231	387.2	387.71	16.07
2500	13896	846.0	846.54	16.41
5000	22107	974.7	975.23	22.67
7500	29268	1068.2	1068.69	27.39
10000	32443	1183.2	1183.76	27.41
15000	39916	1336.5	1337.04	29.85

Table 8.8: *Sequential and parallel execution times for the road network of Pennsylvania*

Northwest USA				
Demand Matrix Size	Sequential	Parallel (execution)	Parallel (total)	Speedup
1000	7555	400.5	401.06	18.84
2500	14496	851.5	852.01	17.01
5000	23364	992.5	992.99	23.53
7500	31405	1073.1	1073.64	29.25
10000	34654	1184.9	1185.42	29.23
15000	40674	1352.1	1352.67	30.07

Table 8.9: *Sequential and parallel execution times for the road network of Northwest USA*

Texas				
Demand Matrix Size	Sequential	Parallel (execution)	Parallel (total)	Speedup
1000	8606	405.8	406.37	21.18
2500	16598	854.8	855.30	19.41
5000	26777	1000.1	1000.58	26.76
7500	35955	1073.8	1074.35	33.47
10000	39689	1190.7	1191.26	33.32
15000	46602	1358.3	1358.79	34.30

Table 8.10: *Sequential and parallel execution times for the road network of Texas*

Northeast USA				
Demand Matrix Size	Sequential	Parallel (execution)	Parallel (total)	Speedup
1000	8197	411.1	411.58	19.92
2500	16097	855.2	855.71	18.81
5000	26181	1003.8	1004.31	26.07
7500	31259	1079.8	1080.37	28.93
10000	34727	1195.8	1196.35	29.03
15000	41398	1358.9	1359.44	30.45

Table 8.11: *Sequential and parallel execution times for the road network of Northeast USA*

California and Nevada				
Demand Matrix Size	Sequential	Parallel (execution)	Parallel (total)	Speedup
1000	10133	412.6	413.11	24.53
2500	19952	862.2	862.70	23.13
5000	32473	1010.2	1010.73	32.13
7500	38709	1080.0	1080.48	35.83
10000	43101	1204.6	1205.09	35.77
15000	51349	1365.3	1365.80	37.60

Table 8.12: *Sequential and parallel execution times for the road network of California and Nevada*

California				
Demand Matrix Size	Sequential	Parallel (execution)	Parallel (total)	Speedup
1000	10598	414.1	414.64	25.56
2500	20933	867.2	867.71	24.12
5000	34154	1017.3	1017.84	33.56
7500	40684	1080.2	1080.74	37.64
10000	45280	1209.7	1210.24	37.41
15000	53989	1365.3	1365.80	39.53

Table 8.13: *Sequential and parallel execution times for the road network of California*

8.4.1 Analysis of the Results

The experiments, carried out in ten road networks of substantial size of USA, have demonstrated that the set of strategies presented throughout this work is capable of performing large scale simulations in a reasonable computational time, using equipment of relatively low cost and easy access.

The experiments also showed the great scalability of GPU algorithms. As the size of the problem to be solved increases, the speedup raises too, with efficiency gains in runtime varying from 9 to 39 times. This shows that GPUs are better equipped than CPUs when it comes to solve parallelizable, large scale problems.

One point to be noted is that, despite the large size of the networks used in the tests, the communication costs between the two main devices involved (CPU and GPU) did not constitute an obstacle to the use of the proposed approach, since even in the largest road network these costs did not exceed 1 second.

The urban networks in these experiments are typical representatives of large meshes. Although all of them relate to North America, there is no reason to believe that

their general structure is substantially different from the traffic networks of other regions of the world, which opens space for the broad use of the strategies described here.

8.5 General Remarks

The current chapter showed how a macroscopic traffic assignment process based on Beckmann's model can be efficiently performed on GPUs. Firstly, a sequential implementation of that process was developed, tested and analyzed in order to identify the most time consuming steps or subroutines.

Then, a GPU-based parallel implementation was presented, which makes intensive use of the parallelism strategies and algorithms discussed in the previous chapters. Computational experiments with both codes demonstrated the effectiveness and efficiency of the GPU algorithm implementation in the context of large road networks.

To the best of our knowledge, the present work is the first to propose a complete parallel implementation of the macroscopic TAP for GPU architectures.

This opens up the possibility of dealing with large-scale simulations by the urban traffic modeling system under development by the Federal University of Goiás (UFG) through its Informatics Institute (INF-UFG) and its Mathematics and Statistics Institute (IME-UFG) called PET-Gyn.

CHAPTER 9

Conclusions

The present thesis studied the use of parallel computing on GPUs for the analysis and simulation of urban traffic and related problems. This chapter summarizes the main contributions of the work and presents ideas for further developments in this area.

The main focus of the thesis was on the proposal, development, implementation and evaluation of parallel algorithms in GPU to solve two well studied problems in the literature: the *Enumeration of Chordless Cycles (holes)* in graphs and the macroscopic *Traffic Assignment Problem (TAP)*.

For each of these problems several difficulties had to be overcome, like specific limitations of the execution environment (GPU hardware) or the inherent challenges when dealing with parallel programming, such as concurrent access and load balancing, among others.

Regarding the enumeration of chordless cycles, the experiments showed mixed results, proving that the performance of the parallel algorithm when compared to its sequential version is closely dependent on the structure of the graph used as input. Graphs with few cycles to be enumerated or low parallelism to be exploited tend to favor the sequential approach, since the CPU is capable of running sequential code more efficiently.

On the other hand, for graphs with a large amount of such structures to be enumerated and/or whose analysis is very complex (those in which the sequential algorithm has a dense and deep recursion tree), there was a clear advantage of the parallel approach proposed.

Furthermore, it should be considered that for an effective use of the GPU capabilities, its hundreds and/or thousands of cores should be busy as much as possible. This is not possible, for example, in the graph represented by Figure 5.5, where only a work-item performs useful work, while all the others are idle. This same phenomenon occurs during the early stages of the enumeration process, where there are few chordless paths to be analyzed.

Regarding the traffic assignment, two computational problems that have to be solved as sub-tasks were tackled first. One problem was the parallel reduction. For it, some improvements have been proposed for already existing solutions. The second problem

was the computation of shortest paths. For that, a GPU-based algorithm for SSSP that works very well for urban traffic networks was devised. Finally, a GPU TAP algorithm was designed, gathering all strategies and approaches developed up to now.

Experiments with the algorithm showed that, as the problem size grows, its speedups also increase over its sequencial equivalent, making possible the simulation of large urban traffic networks.

All stages of this work suggested and reinforced something that is already a consensus in the area: GPUs were built to solve large and complex problems, not being suitable for the resolution and/or reduction in execution time of problems considered simple or trivial.

9.1 Future Work

Although the results presented by the developed parallel algorithms are quite interesting, several future works can be foreseen and developed from this research, which are related to the limitations identified during the development of the present thesis, but that could not be solved or addressed. Next, some of these limitations are briefly discussed:

- For parallelism in all stages of chordless cycles enumeration, the initial task, which performs vertex labeling, should not be done sequentially. Although for the problem at hand this step consumes very little time and does not interfere with the overall performace of the algorithm, especially taking into account the graphs used in the experiments, this technique, once parallelized, can help in a faster resolution of some other problems that heavily depend on it;
- The memory size of current GPUs is a limiting factor for any enumeration algorithm that has to be executed on those devices, even using compact data structures for graph and solution representation. Hence, a new data transportation protocol between the ordinary CPU memory – and some other types of larger memories, if necessary – and the GPU memory has to be developed in order to open space when it is needed, therefore allowing the enumeration process for much larger graphs. Another strategy would be the use of NVLink technology [92];
- The observed behavior in Dijkstra's priority queue $\underline{\boldsymbol{Q}}$ when applied to graphs representing road networks needs a formal mathematical proof that explains this behavior. The present thesis empirically observed this phenomenon and used it as basis for the entire allocation of $\underline{\boldsymbol{Q}}$ in GPU's local memory, but its demonstration is still needed;

Bibliography

[1] ABRASH, M. **Michael Abrash's Graphics Programming Black Book, Special Edition, The Coriolis Group**. *Inc., Arizona*, 1997.

[2] AGARWAL, P.; DUTTA, M. **New approach of Bellman Ford algorithm on GPU using Compute Unified Design Architecture (CUDA)**. *International Journal of Computer Applications*, 110(13), 2015.

[3] AKL, S. G.; GUENTHER, G. R. **Broadcasting with Selective Reduction.** In: *IFIP Congress'89*, p. 515–520, 1989.

[4] AMDAHL, G. M. **Validity of the single processor approach to achieving large scale computing capabilities**. In: *Proceedings of the April 18-20, 1967, spring joint computer conference*, AFIPS '67 (Spring), p. 483–485, New York, NY, USA, 1967. ACM.

[5] ARAGÓN, F. R. C.; LEAL, J. E. **Alocação de fluxos de passageiros em uma rede de transporte público de grande porte formulado como um problema de inequações variacionais**. *Pesquisa Operacional*, 23(2):235–264, Aug. 2003.

[6] ARCHER, J. **Developing the Potential of Micro-Simulation Modelling for Traffic Safety Assessment**. *13th International Cooperation on Theories and Concepts in Traffic Safety - ICTCT - workshop, Corfu*, 2000.

[7] AREZKI, Y. V. V. D. **A Full Analytical Implementation of the PARTAN/Frank-Wolf Algorithm for Equilibrium Assignment**. *Transportation Science*, 1(24):58–62, 1990.

[8] BAR-GERA, H. **Traffic assignment by paired alternative segments**. *Transportation Research Part B: Methodological*, 44(8-9):1022–1046, 2010.

[9] BAUER, R.; DELLING, D.; SANDERS, P.; SCHIEFERDECKER, D.; SCHULTES, D.; WAGNER, D. **Combining Hierarchical and Goal-Directed Speed-up Techniques for Dijkstra's Algorithm**. *Journal of Experimental Algorithmics*, 15:2.3:2.1–2.3:2.31, Mar. 2010.

[10] BAZARAA, M. S.; SHERALI, H. D.; SHETTY, C. M. **Nonlinear Programming: Theory and Algorithms**. John Wiley & Sons, 2013.

[11] BECKMANN, M.; MCGUIRE, C.; WINSTEN, C. **Studies in the Economics of Transportation**. Yale University Press, New Haven, Connecticut, 1956.

[12] BELLMAN, R. **On a routing problem**. *Quarterly of Applied Mathematics*, p. 87–90, 1958.

[13] BERGOMI, M. **Traffic Assignment Problem – The Stochastic User Equilibrium**. Master's thesis, ETH Zurich, Aug. 2009.

[14] BERTSEKAS, D. P. **Nonlinear Programming**. Athena Scientific, 2 edition, 1999.

[15] BERTSEKAS, D. P.; GALLAGER, R. G. **Data Networks**. Prentice-hall, 1987.

[16] BILLETER, M.; OLSSON, O.; ASSARSSON, U. **Efficient Stream Compaction on Wide SIMD Many-Core Architectures**. In: *Proceedings of the Conference on High Performance Graphics 2009*, HPG '09, p. 159–166, New York, NY, USA, 2009. ACM.

[17] BIRMELÉ, E.; FERREIRA, R.; GROSSI, R.; MARINO, A.; PISANTI, N.; RIZZI, R.; SACOMOTO, G. **Optimal Listing of Cycles and st-Paths in Undirected Graphs**. In: *Proceedings of SODA'13 – Annual ACM-SIAM Symposium on Discrete Algorithms*, p. 1884–1896. SIAM, 2013.

[18] BISDORFF, R. **On Enumerating Chordless Circuits in Directed Graphs**, 2010. Available at http://sma.uni.lu/bisdorff/ChordlessCircuits/documents/chordlessCircuits.pdf.

[19] BLOY, K. **An Investigation into Some Aspects of Braess Paradox**. Technical report, Vela VKE Consulting Engineers, Oct. 2006.

[20] BLUM, A.; CHAWLA, S. **Learning from labeled and unlabeled data using graph mincuts**. In: *Proceedings of the Eighteenth International Conference on Machine Learning*, ICML '01, p. 19–26, San Francisco, CA, USA, 2001. Morgan Kaufmann Publishers Inc.

[21] BOGDANOV, A.; TREVISAN, L. **Average-case complexity**. *Found. Trends Theor. Comput. Sci.*, 2(1):1–106, Oct. 2006.

[22] BONABEAU, E. **Agent-based modeling: Methods and techniques for simulating human systems**. *Proceedings of the National Academy of Sciences*, 99(suppl 3):7280–7287, 2002.

[23] BONDY, J. A.; MURTY, U. S. R. **Graph theory with applications**, volume 6. Macmillan London, 1976.

[24] BOXILL, S. A.; YU, L. **An Evaluation of Traffic Simulation Models for Supporting ITS Development**. Technical report, Texas Southern University, Oct. 2000.

[25] BOYCE, D. E.; MAHMASSANI, H. S.; NAGURNEY, A. **A retrospective on Beckmann, McGuire and Winsten's Studies in the Economics of Transportation**. *Papers in Regional Science*, 84(1):85–103, 2005.

[26] BRENT, R. P. **The Parallel Evaluation of General Arithmetic Expressions**. *J. ACM*, 21:201–206, April 1974.

[27] BRODAL, G. S. **Worst-case efficient priority queues.** In: *SODA*, volume 96, p. 52–58, 1996.

[28] BRODAL, G. S.; LAGOGIANNIS, G.; TARJAN, R. E. **Strict Fibonacci heaps**. In: *Proceedings of the Forty-fourth Annual ACM Symposium on Theory of Computing*, STOC '12, p. 1177–1184, New York, NY, USA, 2012. ACM.

[29] BRODAL, G. S.; TRÄFF, J. L.; ZAROLIAGIS, C. D. **A parallel priority queue with constant time operations**. *Journal of Parallel and Distributed Computing*, 49(1):4–21, 1998.

[30] BRODAL, G. S.; TRAFF, J.; ZAROLIAGIS, C. D. **A parallel priority data structure with applications**. In: *Parallel Processing Symposium, 1997. Proceedings., 11th International*, p. 689–693. IEEE, 1997.

[31] BRUYNOOGHE, M.; GIBERT, A.; SAKOROVITCH, M. **Une methode dáffectation du traffic**. In: *Fourth International Symposium on the Theory of Traffic Flow, Karlsruhe*, 1968.

[32] BURGHOUT, W.; KOUTSOPOULOS, H.; ANDREASSON, I. **A discrete-event mesoscopic traffic simulation model for hybrid traffic simulation**. In: *Intelligent Transportation Systems Conference, 2006. ITSC '06. IEEE*, p. 1102–1107, sept. 2006.

[33] BURGHOUT, W.; , J. W. **Hybrid Traffic Simulation with Adaptive Signal Control**. *Transportation Research Record: Journal of the Transportation Research Board*, 1999:191–197, 2007.

[34] BURGHOUT, W. **Hybrid Microscopic-Mesoscopic Traffic Simulation**. PhD thesis, Royal Institute of Technology, Stockholm, Sweden, 2004.

[35] BURGHOUT, W.; KOUTSOPOULOS, H. N. **Hybrid traffic simulation models**. In: Chung, E.; Dumont, A.-G., editors, *Transport Simulation – Beyond Traditional Approaches*, chapter 2. 2009.

[36] CALIXTO, I. C. A. C. **Proposta de um Método de Estimação de Matrizes Origem-Destino Baseado em Programação Linear Fuzzy para Redes Viárias Brasileiras Congestionadas**. Master's thesis, Instituto de Informática – Universidade Federal de Goiás, July 2011.

[37] CAMPBELL, D. K. **A survey of models of parallel computation**. *Report-University of York Department of Computer Science YCS*, 1997.

[38] CASCETTA, E. **Transportation Systems Engineering: Theory and Methods**. Springer, 2001.

[39] CATANZARO, B. **OpenCL Optimization Case Study: Simple Reductions**. http://developer.amd.com/resources/documentation-articles/articles-whitepapers/opencl-optimization-case-study-simple-reductions/, Aug. 2010. published by Advanced Micro Devices. Last accessed in January 05, 2014.

[40] CHAKROUN, I.; MEZMAZ, M.; MELAB, N.; BENDJOUDI, A. **Reducing Thread Divergence in a GPU-Accelerated Branch-and-Bound Algorithm**. *Concurrency and Computation: Practice and Experience*, 25(8):1121–1136, 2013.

[41] CHANDRASEKHARAN, N.; LASKSHMANAN, V.; MEDIDI, M. **Efficient Parallel Algorithms for Finding Chordless Cycles in Graphs**. *Parallel Process. Lett.*, 3(2):165–170, 1993.

[42] CHE, S.; BOYER, M.; MENG, J.; TARJAN, D.; SHEAFFER, J. W.; SKADRON, K. **A Performance Study of General-Purpose Applications on Graphics Processors Using CUDA**. *Journal of Parallel and Distributed Computing*, 68(10):1370–1380, 2008. General-Purpose Processing using Graphics Processing Units.

[43] CHERKASSKY, B. V.; GOLDBERG, A. V.; RADZIK, T. **Shortest paths algorithms: Theory and experimental evaluation**. *Mathematical Programming*, 73(2):129–174, 1996.

[44] CHIANG, A. C. **Fundamental Methods of Mathematical Economics**. McGraw-Hill Higher Education, 3 edition, Jan. 1984.

[45] CHOW, C. M. **BroadCasting with Selective Reduction – An Alternative Implementation and New Algorithms**. Master's thesis, Department of Computing and Information Science, Queen's University, Kingston, Ontario, Nov. 1997.

[46] CHRONOPOULOS, A. T.; JOHNSTON, C. M. **A Real-Time Traffic Simulation System**. *Vehicular Technology, IEEE Transactions on*, 47(1):321–331, feb 1998.

[47] CHUDAK, F. A.; ELEUTERIO, V. D. S.; NESTEROV, Y. **Static Traffic Assignment Problem - A comparison between Beckmann (1956) and Nesterov and de Palma (1998) models**. *7th STRC - Swiss Transport Research Conference, Monte Verità, Ascona*, Sep. 2007.

[48] COHEN, E. **Efficient parallel shortest-paths in digraphs with a separator decomposition**. In: *Proceedings of the fifth annual ACM symposium on Parallel algorithms and architectures*, p. 57–67. ACM, 1993.

[49] COHEN, J. **Food Webs and Niche Space**. Princeton University Press, 1978.

[50] COLE, R.; ZAJICEK, O. **The APRAM: Incorporating Asynchrony into the PRAM Model.** In: *SPAA'89 – Proceedings of the First Annual ACM Symposium on Parallel Algorithms and Architectures*, p. 169–178, 1989.

[51] CORMEN, T. H.; LEISERSON, C. E.; RIVEST, R. L.; STEIN, C. **Algoritmos: Teoria e Prática**. Editora Campus, 2 edition, 2002.

[52] COSTODIO, J. **Problema de Equilíbrio em Redes de Transporte. Comparação entre o Método do Gradiente Projetado e o Método das Combinações Convexas**. Master's thesis, Universidade Federal de Santa Catarina, 2003.

[53] CRAUSER, A.; MEHLHORN, K.; MEYER, U.; SANDERS, P. **A parallelization of dijkstra's shortest path algorithm**. In: *International Symposium on Mathematical Foundations of Computer Science*, p. 722–731. Springer, 1998.

[54] CULLER, D.; KARP, R.; PATTERSON, D.; SAHAY, A.; SCHAUSER, K. E.; SANTOS, E.; SUBRAMONIAN, R.; VON EICKEN, T. **LogP: Towards a Realistic Model of Parallel Computation**. In: *Proceedings of the fourth ACM SIGPLAN symposium on Principles and practice of parallel programming*, PPOPP '93, p. 1–12, New York, NY, USA, 1993. ACM.

[55] CULLER, D. E.; GUPTA, A.; SINGH, J. P. **Parallel Computer Architecture: A Hardware/Software Approach**. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1st edition, 1997.

[56] DA SILVA, P. C. M. **Elementos dos Sistemas de Tráfego**. Technical report, Universidade de Brasília - Faculdade de Tecnologia - Departamento de Engenharia Civil e Ambiental - Área de Transportes, Mar. 2001.

[57] DAFERMOS, S. **Traffic equilibrium and variational inequalities**. *Transportation science*, 14(1):42–54, 1980.

[58] DAI, L. **Fast shortest path algorithm for road network and implementation**. *Carleton University School of Computer Science COMP*, 4905, 2005.

[59] DAI, W.; ZHANG, J.; ZHANG, D. **Parallel Simulation of Large-Scale Microscopic Traffic Networks**. In: *Advanced Computer Control (ICACC), 2010 2nd International Conference on*, volume 3, p. 22–28, march 2010.

[60] DAVIDSON, A.; BAXTER, S.; GARLAND, M.; OWENS, J. D. **Work-efficient parallel GPU methods for single-source shortest paths**. In: *Parallel and Distributed Processing Symposium, 2014 IEEE 28th International*, p. 349–359, May 2014.

[61] DAVIS, R. **The ILLIAC IV Processing Element**. *IEEE Transactions on Computers*, 18:800–816, 1969.

[62] DE ARAÚJO, D. R. C. **Comparação das Simulações de Tráfego dos Modelos SATURN e DRACULA**. Master's thesis, Universidade Federal do Rio Grande do Sul, 2003.

[63] DE MENEZES, R. P. **Um Estudo Sobre Modelos de Computação Paralela**. Master's thesis, Departamento de Ciência da Computação – IMECC – UNICAMP, June 1995.

[64] DE OLIVEIRA, J. L.; SILVA, A. C. D.; HALL, B. R. **Planning Brazilian Urban Traffic with a Geographic Application Software**. *Brazilian Symposium in Geoinformatics, Campos do Jordão*, p. 1–8, 2003.

[65] DE PALMA, A.; NESTEROV, Y. **Optimization Formulations and Static Equilibrium in Congested Transportation Networks**. CORE Discussion Papers 1998061, Université catholique de Louvain, Center for Operations Research and Econometrics (CORE), July 1998.

[66] DE PALMA, A.; NESTEROV, Y. **Stable Dynamics Solutions in Transportation Systems**. Core discussion papers, Université catholique de Louvain, Center for Operations Research and Econometrics (CORE), 2000.

[67] DE S. ALENCAR, W.; FOULDS, L. R.; DO NASCIMENTO, H. A. D.; HALL., B. R.; LONGO, H. J. **Uma aproximação linear da demanda elástica de viagens em redes congestionadas de tráfego urbano com custos assimétricos e dados imprecisos**. In: *Anais do XLVI Simpósio Brasileiro de Pesquisa Operacional (XLVI*

SBPO), p. 1800–1811. Sociedade Brasileira de Pesquisa Operacional (SOBRAPO), SOBRAPO, Nov. 2014.

[68] DEFOUR, D.; COLLANGE, S. **Reproducible floating-point atomic addition in data-parallel environment**. In: *Computer Science and Information Systems (FedCSIS), 2015 Federated Conference on*, p. 721–728, Sept 2015.

[69] DEMETRESCU, C.; GOLDBERG, A.; JOHNSON, D. **9th DIMACS implementation challenge–shortest paths (2006)**, 2006.

[70] DHARMADHIKARI, S.; JOGDEO, K. **Multivariate unimodality**. *The Annals of Statistics*, 4(3):607–613, 1976.

[71] DHILLON, I. S. **Co-clustering documents and words using bipartite spectral graph partitioning**. In: *Proceedings of the Seventh ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '01, p. 269–274, New York, NY, USA, 2001. ACM.

[72] DIA, H.; PANWAI, S. **Nanoscopic traffic simulation: Enhanced models of driver behaviour for its and telematics simulations**. In: *INTERNATIONAL SYMPOSIUM ON TRANSPORT SIMULATION, 8TH, 2008, SURFERS PARADISE, QUEENSLAND, AUSTRALIA*, 2008.

[73] DIAS, E. S. **Reconhecimento Polinomial de Álgebras Cluster de Tipo Finito**. PhD thesis, Instituto de Informática – Universidade Federal de Goiás, Sep. 2015.

[74] DIAS, E. S.; CASTONGUAY, D.; LONGO, H. J.; JRADI, W. A. R. **Efficient Enumeration of All Chordless Cycles in Graphs**. *CoRR*, abs/1309.1051, 2013.

[75] DIJKSTRA, E. W. **A note on two problems in connexion with graphs.** *Numerische Mathematik*, 1:269–271, 1959.

[76] DRISCOLL, J. R.; GABOW, H. N.; SHRAIRMAN, R.; TARJAN, R. E. **Relaxed heaps: An alternative to Fibonacci heaps with applications to parallel computation**. *Commun. ACM*, 31(11):1343–1354, Nov. 1988.

[77] D'SOUZA, R. M.; LYSENKO, M.; RAHMANI, K. **SugarScape on Steroids: Simulating Over a Million Agents at Interactive Rates**. *Proceedings of Agent2007*, 2007.

[78] DUARTE, D. C. S. **LIPSTUD – Um Método de Otimização de Fluxo de Tráfego Urbano Baseado em Proibição e Permissão de Conversões**. Master's thesis, Universidade Federal de Goiás, Mar. 2012.

[79] DUMITRIU, I. **On generalized Tribonacci sequences and additive partitions**. *Discrete Mathematics*, 219(1-3):65–83, 2000.

[80] EL-REWINI, H.; ABD-EL-BARR, M. **Advanced Computer Architecture and Parallel Processing**. Wiley Series On Parallel And Distributed Computing. Wiley, 2005.

[81] ERKAN, G. **Language model-based document clustering using random walks**. In: *Proceedings of the Main Conference on Human Language Technology Conference of the North American Chapter of the Association of Computational Linguistics*, HLT-NAACL '06, p. 479–486, Stroudsburg, PA, USA, 2006. Association for Computational Linguistics.

[82] ERKAN, G.; RADEV, D. R. **Lexpagerank: Prestige in multi-document text summarization**. In: *EMNLP – Conference on Empirical Methods in Natural Language Processing*, Barcelona, Spain, 2004.

[83] ERLEMANN, K.; HARTMANN, D. **Parallelization of a Microscopic Traffic Simulation System Using MPI-Java**. K. Gürlebeck and C. Könke, July 2006.

[84] EUZÉBIO, R. M. G. L. **O Custo do Caos – Prejuízo ao Bolso e ao Meio Ambiente – Cidades não Suportam mais o Crescimento da Frota de Veículos**, Jul. 2012. IPEA – Instituto de Pesquisa Econômica Aplicada.

[85] FEITOSA, F. C. C. **Um Estudo Prático para Contagem Volumétrica Automática de Veículos Usando Visão Computacional**. Master's thesis, Universidade Federal de Goiás, sep 2012.

[86] FERREIRA, R.; GROSSI, R.; RIZZI, R.; SACOMOTO, G.; SAGOT, M.-F. **Algorithms - ESA 2014: 22th Annual European Symposium, Wroclaw, Poland, September 8-10, 2014. Proceedings**, chapter Amortized $\tilde{O}(|V|)$-Delay Algorithm for Listing Chordless Cycles in Undirected Graphs, p. 418–429. Springer Berlin Heidelberg, Berlin, Heidelberg, 2014.

[87] FILHO, J. I. D. O. L. **Pós-Avaliação da Previsão de Demanda por Transportes no Município de Fortaleza**. Master's thesis, Universidade Federal do Ceará, 2003.

[88] FLOYD, R. W. **Algorithm 97: Shortest path**. *Commun. ACM*, 5(6):345–, June 1962.

[89] FLYNN, M. J. **Some computer organizations and their effectiveness**. *IEEE Trans. Comput.*, 21(9):948–960, Sept. 1972.

[90] FOG, A. **The microarchitecture of Intel, AMD and VIA CPUs**. *An optimization guide for assembly programmers and compiler makers. Copenhagen University College of Engineering*, 2011.

[91] FOG, A. **Optimizing Subroutines in Assembly Language: An Optimization Guide for x86 Platforms**. Technical University of Denmark, 2013.

[92] FOLEY, D. **NVLink, Pascal and stacked memory: Feeding the appetite for big data**. *Nvidia.com*, 2014.

[93] FORTUNE, S.; WYLLIE, J. **Parallelism in Random Access Machines**. In: *Proceedings of the tenth annual ACM symposium on Theory of computing*, STOC '78, p. 114–118, New York, NY, USA, 1978. ACM.

[94] FOSTER, I. **Designing and Building Parallel Programs: Concepts and Tools for Parallel Software Engineering**. Parallel programming / scientific computing. Addison-Wesley, 1995.

[95] FRANK, M. **The Braess Paradox**. *Mathematical Programming*, 1(20):283–302, Dec. 1981.

[96] FRANK, M.; WOLFE, P. **An algorithm for quadratic programming**. *Naval Research Logistics Quarterly*, 3(1-2):95–110, 1956.

[97] FREDMAN, M. L.; TARJAN, R. E. **Fibonacci heaps and their uses in improved network optimization algorithms**. *J. ACM*, 34(3):596–615, July 1987.

[98] FUNG, W. W. L.; SHAM, I.; YUAN, G.; AAMODT, T. **Dynamic warp formation and scheduling for efficient GPU control flow**. In: *Microarchitecture, 2007. MICRO 2007. 40th Annual IEEE/ACM International Symposium on*, p. 407–420, Dec 2007.

[99] GAIOSO, R. R. A.; JRADI, W. A. R.; PAULA, L. C. M.; DE S. ALENCAR, W.; DO NASCIMENTO, H. A. D.; MARTINS, W. S.; CACERES, E. N. **Paralelização do algoritmo Floyd-Warshall usando GPU**. In: *Anais do XIV Simpósio em Sistemas Computacionais (WSCAD-SSC)*, p. 19–25, Porto de Galinhas, PE, Brazil, oct. 2013. Sociedade Brasileira de Computação, Editora da SBC.

[100] GAIOSO, R. D. R. A.; OTHERS. **Implementações paralelas para os problemas do fecho transitivo e caminho mínimo APSP na GPU**. 2014.

[101] GEISBERGER, R.; SANDERS, P.; SCHULTES, D.; VETTER, C. **Exact routing in large road networks using contraction hierarchies**. *Transportation Science*, 46(3):388–404, 2012.

[102] GIBBONS, P. B. **A More Practical PRAM Model**. In: *Proceedings of the first annual ACM symposium on Parallel algorithms and architectures*, SPAA '89, p. 158–168, New York, NY, USA, 1989. ACM.

[103] GIBBONS, P. B.; MATIAS, Y. **Efficient Low-Contention Parallel Algorithms**. *J. Comput. Syst. Sci.*, 53:417–442, December 1996.

[104] GIBBONS, P. B.; MATIAS, Y.; RAMACHANDRAN, V. **Efficient Low-Contention Parallel Algorithms**. In: *Proceedings of the sixth annual ACM symposium on Parallel algorithms and architectures*, SPAA '94, p. 236–247, New York, NY, USA, 1994. ACM.

[105] GIBBONS, P. B.; MATIAS, Y.; RAMACHANDRAN, V. **The QRQW PRAM: Accounting for Contention in Parallel Algorithms**. In: *Proceedings of the fifth annual ACM-SIAM symposium on Discrete algorithms*, SODA '94, p. 638–648, Philadelphia, PA, USA, 1994. Society for Industrial and Applied Mathematics.

[106] GOBBO, A. F. **Proposta de Aplicação de Sistemas de Inferência Neuro-Fuzzy para Otimização de Tráfego**. Master's thesis, Centro Federal de Educação Tecnológica do Estado do Paraná, Mar. 2005.

[107] GOLDBERG, A. V. **Shortest Path Algorithms: Engineering Aspects**, p. 502–513. Springer Berlin Heidelberg, Berlin, Heidelberg, 2001.

[108] GOLDBERG, A. V. **Point-to-Point Shortest Path Algorithms with Preprocessing**, p. 88–102. Springer Berlin Heidelberg, Berlin, Heidelberg, 2007.

[109] GOLDBERG, D. **What every computer scientist should know about floating-point arithmetic**. *ACM Comput. Surv.*, 23(1):5–48, Mar. 1991.

[110] GOLDMAN, A. **Modelos para a Computação Paralela**. *Escola Regional de Alto Desempenho, Santa Maria*, p. 35 – 66, 2003.

[111] GONDIM, H. W. A. S.; DO NASCIMENTO, H. A. D.; REILLY, D. **Visualizing large scale vehicle traffic network data – a survey of the state-of-the-art**. In: *Information Visualization Theory and Applications (IVAPP), 2014 International Conference on*, p. 337–346, Jan 2014.

[112] GONDIM, H. W. A. S.; NASCIMENTO, H. A. D. D.; REILLY, D. **Visualizações de matrizes origem-destino no cenário do tráfego urbano**. Workshop on Visual Analytics, Information Visualization and Scientific Visualization, 6. (WVIS), aug 2014.

[113] GONDIM, H. W.; NASCIMENTO, H. A. D. D.; REILLY, D. **OD flows - a visual representation of origin-destination matrices in urban traffic scenarios**. Porto Alegre, 2015. Workshop on Visual Analytics, Information Visualization and Scientific Visualization, 6. (WVIS), Sociedade Brasileira de Computação.

[114] GOTZ, A. W.; WILLIAMSON, M. J.; XU, D.; POOLE, D.; GRAND, S. L.; WALKER, R. C. **Routine microsecond molecular dynamics simulations with AMBER on GPUs.** *Journal of Chemical Theory and Computation*, 8(5):1542–1555, 2012. PMID: 22582031.

[115] GRAMA, A.; KARYPIS, G.; KUMAR, V.; GUPTA, A. **Introduction to Parallel Computing (2nd Edition)**. Addison Wesley, 2 edition, January 2003.

[116] GROUP, K. O. W.; OTHERS. **The OpenCL specification**. *version*, 1(29):8, 2008.

[117] GUPTA, K.; STUART, J. A.; OWENS, J. D. **A Study of Persistent Threads Style GPU Programming for GPGPU Workloads**. In: *Innovative Parallel Computing (InPar), 2012*, p. 1–14. IEEE, 2012.

[118] GUSTAFSON, J. L. **Reevaluating Amdahl's Law**. *Commun. ACM*, 31:532–533, May 1988.

[119] HAAS, R.; HOFFMANN, M. **Chordless Paths Through Three Vertices**. *Theor. Comput. Sci.*, 351:360–371, 2006.

[120] HAEUPLER, B.; SEN, S.; TARJAN, R. E. **Rank-pairing heaps**. *SIAM Journal on Computing*, 40(6):1463–1485, 2011.

[121] HAJELA, G.; PANDEY, M. **Parallel implementations for solving shortest path problem using bellman-ford**. *International Journal of Computer Applications*, 95(15), 2014.

[122] HAN, T. D.; ABDELRAHMAN, T. S. **Reducing branch divergence in GPU programs**. In: *Proceedings of the Fourth Workshop on General Purpose Processing on Graphics Processing Units*, GPGPU-4, p. 3:1–3:8, New York, NY, USA, 2011. ACM.

[123] HARISH, P.; NARAYANAN, P. **Accelerating large graph algorithms on the GPU using CUDA**. In: *Proceedings of HiPC '07*, p. 197–208. Springer-Verlag, 2007.

[124] HARRIS, M.; OTHERS. **Optimizing Parallel Reduction in CUDA**. *NVIDIA Developer Technology*, 2, 2007.

[125] HARTENSTEIN, R. **A decade of reconfigurable computing: A visionary retrospective**. In: *Proceedings of the Conference on Design, Automation and Test in Europe*, DATE '01, p. 642–649, Piscataway, NJ, USA, 2001. IEEE Press.

[126] HENNESSY, J.; PATTERSON, D. **Computer Architecture: A Quantitative Approach**. Elsevier, 5th edition, 2011.

[127] HENSHER, D. A.; BUTTON, K. J. **Handbook of Transport Modelling**. Publishing, Emerald Group, 2002.

[128] HEYWOOD, T. H.; RANKA, S. **A practical hierarchical model of parallel computation. The model**. *Journal of Parallel and Distributed Computing*, 16(3):212–232, 1992.

[129] HIGHAM, N. **Accuracy and Stability of Numerical Algorithms: Second Edition**. Society for Industrial and Applied Mathematics, 2002.

[130] HILLIS, W. D. **The Connection Machine**. MIT Press, Cambridge, MA, USA, 1985.

[131] HORD, R. M. **The Illiac IV, The First Supercomputer**. Computer Science Press, Inc., Rockville, MD, USA, 1982.

[132] HOUSTON, M. **Anatomy of AMD's TeraScale Graphics Engine**. http://s08.idav.ucdavis.edu/houston-amd-terascale.pdf, Dec. 2008.

[133] HUANG, J. C.; LENG, T. **Generalized Loop-Unrolling: a Method for Program Speed-Up**. In: *in Proc. IEEE Symp. on Application-Specific Systems and Software Engineering and Technology*, p. 244–248, 1997.

[134] IACONO, J. **Improved Upper Bounds for Pairing Heaps**, p. 32–45. Springer Berlin Heidelberg, Berlin, Heidelberg, 2000.

[135] JÁJÁ, J. **An introduction to parallel algorithms**. Addison Wesley Longman Publishing Co., Inc., Redwood City, CA, USA, 1992.

[136] JEONG, I.-K.; UDDIN, J.; KANG, M.; KIM, C.-H.; KIM, J.-M. **Accelerating a Bellman–Ford routing algorithm using GPU**. In: *Frontier and Innovation in Future Computing and Communications*, p. 153–160. Springer, 2014.

[137] JOACHIMS, T.; OTHERS. **Transductive learning via spectral graph partitioning**. In: *ICML – International Conference on Machine Learning*, volume 3, p. 290–297, 2003.

[138] JOHNSON, D. B. **Efficient algorithms for shortest paths in sparse networks**. *J. ACM*, 24(1):1–13, Jan. 1977.

[139] JOHNSON, D. B. **Priority queues with update and finding minimum spanning trees**. *Information Processing Letters*, 4(3):53 – 57, 1975.

[140] JOHNSON, E. E. **Completing an MIMD Multiprocessor Taxonomy**. *ACM SIGARCH – Special Interest Group on Computer Architecture*, 16(3):44–47, June 1988.

[141] JRADI, W. A. R. **Uma Arquitetura de Software Interativo para Apoio a Decisão na Modelagem e Análise do Tráfego Urbano**. Master's thesis, Universidade Federal de Goiás, Oct. 2008.

[142] JRADI, W. A.; DO NASCIMENT, H. A.; LONGO, H.; HALL, B. R. **Simulation and analysis of urban traffic – the architecture of a web-based interactive decision support system**. In: *2009 12th International IEEE Conference on Intelligent Transportation Systems*, p. 1–6. IEEE, 2009.

[143] KAHAN, W. **Pracniques: Further remarks on reducing truncation errors**. *Commun. ACM*, 8(1):40–, Jan. 1965.

[144] KAHLE, B. A.; HILLIS, W. D. **The Connection Machine model CM-1 architecture**. *IEEE Transactions on Systems, Mans, Cybernetics*, 19(4):707–713, July 1989.

[145] KAMPS, S. **Network holes and traffic congestion**. http://www.geos.ed.ac.uk/~mscgis/05-06/s0565603/, Aug. 2006.

[146] KAPOOR, S.; RAMESH, H. **An Algorithm for Enumerating All Spanning Trees of a Directed Graph**. *Algorithmica*, 27(2):120–130, 2000.

[147] KARP, R. M.; RAMACHANDRAN, V. **Parallel Algorithms for Shared-Memory Machines**, p. 869–941. MIT Press, Cambridge, MA, USA, 1990.

[148] KARP, R. M. **Parallel Combinatorial Computing**. Jan. 1991.

[149] KIEFER, J. C. **Sequential Minimax Search for a Maximum**. *Proc. Am. Math. Soc.*, 4:502–506, 1953.

[150] KLEINBERG, J.; TARDOS, E. **Algorithm Design**. Pearson Education India, 2006.

[151] KNIGHT, F. H. **Some fallacies in the interpretation of social cost**. *The Quarterly Journal of Economics*, p. 582–606, 1924.

[152] KNOPP, S.; SANDERS, P.; SCHULTES, D.; SCHULZ, F.; WAGNER, D. **Computing many-to-many shortest paths using highway hierarchies**. In: *Proceedings of the Meeting on Algorithm Engineering & Expermiments*, p. 36–45, Philadelphia, PA, USA, 2007. Society for Industrial and Applied Mathematics.

[153] KOLATA, G. **What if They Closed 42nd Street and Nobody Noticed?** *The New York Times, Dec. 25*, 1990.

[154] KOSKINEN, K.; KOSONEN, I.; LUTTINEN, T.; SCHIROKOFF, A.; LUOMA, J. **Development of a nanoscopic traffic simulation tool**. *Advances in transportation studies*, 2009(17):89–96, 2009.

[155] KOUTSOUPIAS, E.; PAPADIMITRIOU, C. **Worst-Case Equilibria**. In: *Proceedings of the 16th Annual Conference on Theoretical Aspects of Computer Science*, STACS'99, p. 404–413, Berlin, Heidelberg, 1999. Springer-Verlag.

[156] KUMAR, A.; PEETA, S. **Slope-Based Multipath Flow Update Algorithm for Static User Equilibrium Traffic Assignment Problem**. *Transportation Research Record: Journal of the Transportation Research Board*, 2196:1–10, 2010.

[157] KUMAR, S.; MISRA, A.; TOMAR, R. S. **A modified parallel approach to single source shortest path problem for massively dense graphs using CUDA**. In: *Computer and Communication Technology (ICCCT), 2011 2nd International Conference on*, p. 635–639. IEEE, 2011.

[158] LEBLANC, L. T. **An Algorithm for the Discrete Network Design Problem**. *Transportation Science*, 9(3):183–199, 1975.

[159] LEE, A.; STREINU, I. **Pebble game algorithms and sparse graphs**. *Discrete Mathematics*, 308(8):1425 – 1437, 2008. Third European Conference on Combinatorics, Graph Theory and Applications. Third European Conference on Combinatorics.

[160] LESKOVEC, J.; KREVL, A. **SNAP Datasets: Stanford large network dataset collection**. http://snap.stanford.edu/data, June 2014.

[161] LI, M.; YESHA, Y. **New lower bounds for parallel computation**. In: *Proceedings of the eighteenth annual ACM symposium on Theory of computing*, STOC '86, p. 177–187, New York, NY, USA, 1986. ACM.

[162] LIEBERMAN, E.; RATHI, A. K. **Revised Traffic Flow Theory: A State-of-the-Art Report**, chapter 10. National Academy of Sciences, Transportation Research Board, Committee on Traffic Flow Theory and Characteristics, 2001.

[163] LIEBERMAN, E. B. **Brief history of traffic simulation**. *Traffic and Transportation Simulation*, p. 17, 2014.

[164] LINDHOLM, E.; NICKOLLS, J.; OBERMAN, S.; MONTRYM, J. **NVIDIA Tesla: A unified graphics and computing architecture**. *IEEE Micro*, 28(2):39–55, March 2008.

[165] LIU, H. X.; MA, W. **A Distributed Modelling Framework for Large-Scale Microscopic Traffic Simulation**. *World Review of Intermodal Transportation Research*, 2(2/3):127–126, 2009.

[166] LIU, H. X.; MA, W.; JAYAKRISHNAN, R.; RECKER, W.; LIU, H. X.; MA, W.; JAYAKRISHNAN, R.; RECKER, W. **A Distributed Modeling Framework for Large-Scale Microscopic Traffic Simulation**, 2005.

[167] LUITJENS, J. **Faster Parallel Reductions on Kepler**. *White Paper*, Feb. 2014. published by NVidia Inc. Last accessed in July 25, 2014.

[168] MAKINO, K.; UNO, T. **New Algorithms for Enumerating All Maximal Cliques**. *Lecture Notes in Comput. Sci., SWAT 2004*, 3111:260–272, 2004.

[169] MAMMAR, S.; SMAILI, S.; MAMMAR, S.; WEIDMANN, G. **A Hybrid Model Based on a Generic Second Order Model**. Transportation Research Board, Jan. 2011.

[170] MARQUET, P.; DUQUENNOY, S.; LE BEUX, S.; MEFTALI, S.; DEKEYSER, J.-L. **Massively parallel processing on a chip**. In: *Proceedings of the 4th international conference on Computing frontiers*, CF '07, p. 277–286, New York, NY, USA, 2007. ACM.

[171] MARTÍN, P. J.; TORRES, R.; GAVILANES, A. **Cuda solutions for the sssp problem**. In: *International Conference on Computational Science*, p. 904–913. Springer, 2009.

[172] MCCOOL, M.; REINDERS, J.; ROBISON, A. **Structured Parallel Programming: Patterns for Efficient Computation**. Elsevier Science, 2012.

[173] MENG, J.; TARJAN, D.; SKADRON, K. **Dynamic Warp Subdivision for Integrated Branch and Memory Divergence Tolerance**. *SIGARCH Comput. Archit. News*, 38(3):235–246, June 2010.

[174] MEYER, U.; SANDERS, P. **δ-stepping: A parallelizable shortest path algorithm**. *Journal of Algorithms*, 49(1):114–152, Oct. 2003.

[175] MEYER, U. **Design and Analysis of Sequential and Parallel Single-Source Shortest-Paths Algorithms**. PhD thesis, Universitätsbibliothek, 2002.

[176] MEYER, U. **Average-case complexity of single-source shortest-paths algorithms: Lower and upper bounds**. *Journal of Algorithms*, 48(1):91–134, 2003.

[177] MICROSOFT. **DirectCompute PDC HOL**. Dec. 2009.

[178] MULLER, J.; BRISEBARRE, N.; DE DINECHIN, F.; JEANNEROD, C.; LEFÈVRE, V.; MELQUIOND, G.; REVOL, N.; STEHLÉ, D.; TORRES, S. **Handbook of Floating-Point Arithmetic**. Birkhäuser Boston, 2009.

[179] Muller-Hannemann, M.; Schirra, S., editors. **Algorithm Engineering: Bridging the Gap Between Algorithm Theory and Practice**. Springer-Verlag, Berlin, Heidelberg, 2010.

[180] MUNSHI, A.; GASTER, B.; MATTSON, T. G.; GINSBURG, D. **OpenCL programming guide**. Pearson Education, 2011.

[181] MURCHLAND, J. D. **Road network traffic distribution in equilibrium**. In: *Proceedings of the conference mathematical methods in economic sciences. Oberwolfach W. Germany, Mathematisches Forschungsinstitut*, p. 145–183, 1969.

[182] NAGURNEY, A. **Network Economics: A Variational Inequality Approach**. Advances in Computational Economics (Book 10). Springer, 2 edition, Dec. 1998.

[183] NARASIMAN, V.; SHEBANOW, M.; LEE, C. J.; MIFTAKHUTDINOV, R.; MUTLU, O.; PATT, Y. N. **Improving GPU Performance via Large Warps and Two-level Warp Scheduling**. In: *Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-44, p. 308–317, New York, NY, USA, 2011. ACM.

[184] NASRE, R.; BURTSCHER, M.; PINGALI, K. **Data-driven versus topology-driven irregular computations on GPUs**. In: *Parallel Distributed Processing (IPDPS), 2013 IEEE 27th International Symposium on*, p. 463–474, 2013.

[185] NAZARI, S.; MEYBODI, M. R.; SALEHIGH, M. A.; TAGHIPOUR, S. **An advanced algorithm for finding shortest path in car navigation system**. In: *Intelligent Networks and Intelligent Systems, 2008. ICINIS '08. First International Conference on*, p. 671–674, Nov 2008.

[186] NEŠETRIL, J.; OSSONA DE MENDEZ, P. **From sparse graphs to nowhere dense structures: Decompositions, independence, dualities and limits**. In: *European Congress of Mathematics*, p. 135–165, 2009.

[187] NESTEROV, Y.; DE PALMA, A. **Stationary Dynamic Solutions in Congested Transportation Networks: Summary and Perspectives**. *Networks and Spatial Economics*, 3:371–395, 2003.

[188] NEVES, P. T. **Variações e aplicações do algoritmo de Dijkstra**. Master's thesis, Instituto de Computação – Universidade de Campinas – UNICAMP, Aug. 2007.

[189] NOVAES, A. G. **Métodos de Otimização: Aplicação aos Transportes**. *Editora Edgard Blücher*, 1978.

[190] ORTEGA-ARRANZ, H.; TORRES, Y.; LLANOS, D.; GONZALEZ-ESCRIBANO, A. **A new gpu-based approach to the shortest path problem**. In: *High performance computing and simulation (HPCS), 2013 international Conference on*, p. 505–511. IEEE, 2013.

[191] ORTEGA-ARRANZ, H.; TORRES, Y.; LLANOS, D. R.; GONZALEZ-ESCRIBANO, A. **The all-pair shortest-path problem in shared-memory heterogeneous systems**. *High-Performance Computing on Complex Environments*, p. 283–299, 2013.

[192] ORTÚZAR, J. D. D.; WILLUMSEN, L. G. **Modelling Transport**. John Wiley and Sons, 3rd edition, 2001.

[193] P. E. HART, N. J. N.; RAPHAEL, B. **A formal basis for the heuristic determination of minimum cost paths**. *IEEE Transactions on Systems, Science, and Cybernetics*, SSC-4(2):100–107, 1968.

[194] PAPAEFTHYMIOU, M.; RODRIGUE, J. **Implementing parallel shortest-paths algorithms**. *DIMACS Series in Discrete Mathematics and Theoretical Computer Science*, 30:59–68, 1997.

[195] PARHAMI, B. **Introduction to Parallel Processing: Algorithms and Architectures**. Plenum series in computer science. Plenum Press, 1999.

[196] PASSOS, L. S.; ROSSETTI, R. J.; KOKKINOGENIS, Z. **Towards the next-generation traffic simulation tools: A first appraisal**. In: *Information Systems and Technologies (CISTI), 2011 6th Iberian Conference on*, p. 1–6. IEEE, 2011.

[197] PATRIKSSON, M. **The Traffic Assignment Problem: Models and Methods**. Topics in Transportation, VSP, Utrecht, The Netherlands, 1994.

[198] PEVZNER, P. A.; TANG, H.; WATERMAN, M. S. **An Eulerian path approach to DNA fragment assembly**. *Proceedings of the National Academy of Sciences*, 98(17):9748–9753, 2001.

[199] PIGNATARO, L. J. **Traffic Engineering: Theory and Practice**. Prentice-Hall. Englewood Cliffs, EUA, 1973.

[200] PINTO, A. B.; DIÓGENES, M. C.; LINDAU, L. A. **Quantificação dos Impactos de Pólos Geradores de Tráfego**. *Universidade Federal do Rio Grande do Sul – UFRGS, Rio Grande do Sul*, 2003.

[201] POYARES, C. N. **Critérios para análise dos efeitos de políticas de restrição ao uso de automóveis em áreas centrais**. Master's thesis, Universidade Federal do Rio de Janeiro, 2000.

[202] PRADHAN, A.; MAHINTHAKUMAR, G. **Finding all-pairs shortest path for a large-scale transportation network using parallel Floyd-Warshall and parallel Dijkstra algorithms**. *Journal of Computing in Civil Engineering*, 27(3):263–273, 2012.

[203] QUINN, M. J.; METOYER, R. A.; HUNTER-ZAWORSKI, K. **Parallel Implementation of the Social Forces Model**. In: *in Proceedings of the Second International Conference in Pedestrian and Evacuation Dynamics*, p. 63–74, 2003.

[204] RAJASEKARAN, S.; REIF, J. **Handbook of Parallel Computing: Models, Algorithms and Applications; Electronic Version**. Chapman and Hall/CRC Computer and Information Science Series. Taylor and Francis Ltd, Hoboken, NJ, 2007.

[205] RAMAN, V.; SANKAR, P.; KUMAR, S.; ASOKAN, K.; RAJ, M. **Analysis of road network of the buffer area of Kochi metro rail service using tools of social network analysis**. *International Conference on Information Science*, July 2014.

[206] RATROUT, N. T.; RAHMAN, S. M. **A Comparative Analysis of Currently Used Microscopic and Macroscopic Traffic Simulation Software**. *The Arabian Journal for Science and Engineering, Number 1B*, 34:121–133, Apr. 2009.

[207] READ, R.; TARJAN, R. **Bounds on Backtrack Algorithms for Listing Cycles, Paths and Spanning Trees**. *Networks*, 5:237–252, 1975.

[208] REITSMA, F.; ENGEL, S. **Searching for 2d spatial network holes**. In: *International Conference on Computational Science and Its Applications*, p. 1069–1078. Springer, 2004.

[209] RIORDAN, O.; WORMALD, N. **The diameter of sparse random graphs**. *Combinatorics, Probability and Computing*, 19(5-6):835–926, 2010.

[210] ROS, F. J.; MARTINEZ, J. A.; RUIZ, P. M. **A survey on modeling and simulation of vehicular networks: Communications, mobility, and tools**. *Computer Communications*, 43:1–15, 2014.

[211] ROSEN, J. B. **The gradient projection method for nonlinear programming. Part I. Linear constraints**. *Journal of the Society for Industrial and Applied Mathematics*, 8(1):181–217, 1960.

[212] ROY, K. **Optimum gate ordering of cmos logic gates using euler path approach: Some insights and explanations.** *CIT – Journal of Computing and Information Technology*, 15(1):85–92, 2007.

[213] SALOMON-FERRER, R.; CASE, D. A.; WALKER, R. C. **An overview of the AMBER biomolecular simulation package**. *Wiley Interdisciplinary Reviews: Computational Molecular Science*, 3(2):198–210, 2013.

[214] SALOMON-FERRER, R.; GÖTZ, A. W.; POOLE, D.; GRAND, S. L.; WALKER, R. C. **Routine microsecond molecular dynamics simulations with AMBER on GPUs.** *Journal of Chemical Theory and Computation*, 9(9):3878–3888, 2013.

[215] SANDERS, P.; SCHULTES, D. **Highway Hierarchies Hasten Exact Shortest Path Queries**, p. 568–579. Springer Berlin Heidelberg, Berlin, Heidelberg, 2005.

[216] SANDERS, P.; SCHULTES, D. **Engineering Highway Hierarchies**, p. 804–816. Springer Berlin Heidelberg, Berlin, Heidelberg, 2006.

[217] SANKAR, K.; SARAD, A. V. **A Time and Memory Efficient Way to Enumerate Cycles in a Graph**. In: *Proceedings of ICIAS – International Conference on Intelligent and Advanced Systems*, p. 498–500. IEEE, 2007.

[218] SANO, Y.; FUKUTA, N. **A GPU-based framework for large-scale multi-agent traffic simulations**. In: *Advanced Applied Informatics (IIAIAAI), 2013 IIAI International Conference on*, p. 262–267, Aug 2013.

[219] SARKAR, V. **Optimized Unrolling of Nested Loops**. *Int. J. Parallel Program.*, 29(5):545–581, Oct. 2001.

[220] SATOH, H.; KOSHINO, H.; UNO, T.; KOICHI, S.; IWATA, S.; NAKATA, T. **Effective consideration of ring structures in CAST/CNMR for highly accurate 13C {NMR} chemical shift prediction**. *Tetrahedron*, 61(31):7431–7437, 2005.

[221] SAUDI, A. B. **Parallel Computing – Lecture Notes**. Technical report, Universiti Malaysia Sabah, Apr. 2008.

[222] SAUMTALLY, T.; LEBACQUE, J.-P.; HAJ-SALEM, H. **Static Traffic Assignment with Side Constraints in a Dense Orthotropic Network**. *Procedia - Social and Behavioral Sciences*, 20(0):465–474, 2011.

[223] SCHEUTZ, M.; SCHERMERHORN, P. **Adaptive algorithms for the dynamic distribution and parallel execution of agent-based models**. *J. Parallel Distrib. Comput.*, 66:1037–1051, August 2006.

[224] SCHULZ, F.; WAGNER, D.; WEIHE, K. **Dijkstra's Algorithm On-Line: An Empirical Case Study from Public Railroad Transport**, p. 110–123. Springer Berlin Heidelberg, Berlin, Heidelberg, 1999.

[225] SHEFFI, Y. **Urban Transportation Networks**. Prentice-Hall, Englewood Cliffs, NJ, 1985.

[226] SHEN, Z.; WANG, K.; ZHU, F. **Agent-based traffic simulation and traffic signal timing optimization with GPU**. In: *2011 14th International IEEE Conference on Intelligent Transportation Systems (ITSC)*, p. 145–150, Oct 2011.

[227] SILBERSCHATZ, A.; GALVIN, P. B.; GAGNE, G. **Operating System Concepts**. Wiley Publishing, 9th edition, 2013.

[228] SINGH, D. P.; KHARE, N. **A study of different parallel implementations of single source shortest path algorithms**. *International Journal of Computer Applications*, 54(10):26–30, September 2012.

[229] SOKHN, N.; BALTENSPERGER, R.; BERSIER, L.-F.; HENNEBERT, J.; ULTES-NITSCHE, U. **Identification of chordless cycle in ecological networks**. In: *International Conference on Complex Sciences*, p. 316–324. Springer, 2012.

[230] STEINBERGER, M.; KAINZ, B.; KERBL, B.; HAUSWIESNER, S.; KENZEL, M.; SCHMALSTIEG, D. **Softshell: Dynamic scheduling on GPUs**. *ACM Trans. Graph.*, 31(6):161:1–161:11, Nov. 2012.

[231] TANG, Y.; ZHANG, Y.; CHEN, H. **A parallel shortest path algorithm based on graph-partitioning and iterative correcting**. In: *High Performance Computing and Communications, 2008. HPCC'08. 10th IEEE International Conference on*, p. 155–161. IEEE, 2008.

[232] TARJAN, R. E. **Enumeration of the Elementary Circuits of a Directed Graph**. *SIAM J. Comput.*, 2(3):211–216, 1973.

[233] THORUP, M. **Undirected single source shortest paths in linear time**. In: *Foundations of Computer Science, 1997. Proceedings., 38th Annual Symposium on*, p. 12–21, Oct 1997.

[234] THORUP, M. **Undirected single-source shortest paths with positive integer weights in linear time**. *J. ACM*, 46(3):362–394, May 1999.

[235] TOLFO, J. D. **Estudo Comparativo de Técnicas de Análise de Desempenho de Redes Viárias no Entorno de Pólos Geradores de Viagens**. Master's thesis, COPPE/UFRJ, Feb. 2006.

[236] TOMITA, E.; TANAKA, A.; TAKAHASHI, H. **The Worst-case Time Complexity for Generating All Maximal Cliques and Computational Experiments**. *Theo. Comp. Sci.*, 363:28–42, 2006.

[237] TRÄFF, J. L.; ZAROLIAGIS, C. D. **A simple parallel algorithm for the single-source shortest path problem on planar digraphs**. In: *International Workshop on Parallel Algorithms for Irregularly Structured Problems*, p. 183–194. Springer, 1996.

[238] TSUCHIYAMA, R.; NAKAMURA, T.; IIZUKA, T.; ASAHARA, A.; MIKI, S. **The OpenCL Programming Book**. Fixstars Corporation, 2010.

[239] UNO, T.; SATOH, H. **An efficient algorithm for enumerating chordless cycles and chordless paths**. In: *Discovery Science*, p. 313–324. Springer, 2014.

[240] VALIANT, L. **The Complexity of Enumeration and Reliability Problems**. *SIAM Journal on Computing*, 8(3):410–421, 1979.

[241] VAN, A. L. **STAQ – Static Traffic Assignment with Queuing**, March 2011.

[242] VILARÓ, J. C.; TORDAY, A.; GERODIMOS, A. **Combining Mesoscopic and Microscopic Simulation in an Integrated Environment as a Hybrid Solution**. *Intelligent Transportation Systems Magazine, IEEE*, 2(3):25–33, fall 2010.

[243] VON NEUMANN, J.; GODFREY, M. D. **First draft of a report on the edvac**. *IEEE Annals of the History of Computing*, 15(4):27–75, 1993.

[244] VUILLEMIN, J. **A data structure for manipulating priority queues**. *Commun. ACM*, 21(4):309–315, Apr. 1978.

[245] WAGNER, D.; WILLHALM, T. **Geometric Speed-Up Techniques for Finding Shortest Paths in Large Sparse Graphs**, p. 776–787. Springer Berlin Heidelberg, Berlin, Heidelberg, 2003.

[246] WANG, L.; MAO, B.; CHEN, S.; ZHANG, K. **A P2P Computational Grid-Based Parallel Traffic Micro-Simulation Model for Large Scale Transportation Networks**. *Computational Sciences and Optimization, International Joint Conference on*, 2:95–99, 2009.

[247] WARDROP, J. G. **Some Theoretical Aspects of Road Traffic Research**, volume 1. Proceedings of Institute of Civil Engineers, 1952.

[248] WEI, D.; CHEN, F.; SUN, X. **An Improved Road Network Partition Algorithm for Parallel Microscopic Traffic Simulation**. In: *Mechanic Automation and Control Engineering (MACE), 2010 International Conference on*, p. 2777–2782, june 2010.

[249] WEST, D. B.; OTHERS. **Introduction to Graph Theory**, volume 2. Prentice hall Upper Saddle River, 2001.

[250] WILD, M. **Generating all Cycles, Chordless Cycles, and Hamiltonian Cycles with the Principle of Exclusion**. *J. Discrete Algorithms*, 6(1):93–102, 2008.

[251] WILSON, R. J.; WATKINS, J. J. **Graphs: An Introductory Approach**. Wiley, Michigan University, 1990.

[252] WILT, N. **The CUDA Handbook: A Comprehensive Guide to GPU Programming**. Pearson Education, 2013.

[253] XIAO, S.; FENG, W.-C. **Inter-block GPU communication via fast barrier synchronization**. In: *Parallel & Distributed Processing (IPDPS), 2010 IEEE International Symposium on*, p. 1–12. IEEE, 2010.

[254] YANG, Q.; MORGAN, D. **Hybrid Traffic Simulation Model**. Transportation Research Board, Jan. 2006.

[255] ZHAN, F. B.; NOON, C. E. **A comparison between label-setting and label-correcting algorithms for computing one-to-one shortest paths**. *Journal of Geographic information and decision analysis*, 4(2):1–11, 2000.

[256] ZHANG, E. Z.; JIANG, Y.; GUO, Z.; SHEN, X. **Streamlining GPU Applications on the Fly: Thread Divergence Elimination Through Runtime Thread-data Remapping**. In: *Proceedings of the 24th ACM International Conference on Supercomputing*, ICS '10, p. 115–126, New York, NY, USA, 2010. ACM.

APPENDIX A

Parallel Computing Models

As previously mentioned (Section 3.1), there are many parallel execution models. One of the mostly used model is the PRAM, described next with further details.

A.1 The PRAM Model

The *Parallel Random Machine* was the first proposed model of parallel computation [93] which enabled the development of algorithms using conventional data structures, defined as being a parallel extension of RAM model. A PRAM consists of an unlimited number of identical processors $P_0, P_1, \ldots$, and also an unlimited global memory – used both for data storage and for communication between processors –, a set of input records and a finite program. Each processor has a not limited local memory and some other attributes, operating synchronously with the other processors.

Although this model is unrealistic in practice [110, 63, 135, 204], so far it remains a reference in the analysis and comparison of parallel algorithms [147] due to its several advantages:

- ***High level of abstraction***, allowing the developer to focus on the structure of the problem and not worrying about details not related to the topic under study (for example, low-level details like hardware architecture, network topology, etc.). Every effort is, therefore, concentrated on the development of efficient algorithms, independently of the execution platform;
- ***Complexity analysis*** very similar to the analysis of algorithms in the RAM model, as only a few additional informations are used related to parallelism. Similarly to the RAM model, complexity in PRAM is expressed using asymptotic notation.
- ***Use of previously acquired skills*** in the development of sequential algorithms. Thanks to the simplicity of PRAM, Richard M. Karp suggests, in Section 6 of his article "Parallel Combinatorial Computing" [148] that all development of parallel algorithms should be divided into two phases: the first consisting of the creation of PRAM algorithm itself and the second to adapt it to the real target platform. This

process is presented as being more appropriate than to develop directly for the real machine.

Among its drawbacks, the following points may be mentioned [63]:

- ***Currently it's not possible to conceive a machine with processors and memory growing indefinitely***. The PRAM model is based on no limitation in the amount of available memory for programs and data. Although today the 64-bit processors are already a reality, being able to manage a huge amount of memory, in practice this memory is not available in sufficient volume, especially in tasks that require the handling of a very large volume of data;
- ***The model ignores the complexity of communication between processors***. Factors such as bandwidth, delays and overheads in communication lead to quite different costs of algorithms, depending on the hardware architecture in use. These costs are ignored in the PRAM model, but are taken into consideration in LogP [54].
- ***Unrealistic assumption that all instructions have unity cost***. This assumption is clearly not applicable in real machines: even the simplest primitive operations, such as addition or multiplication, have completely different costs. Because of this phenomenon, it is entirely possible that two algorithms developed under the PRAM model, A and B, when implemented in real machines present a very different performance from that originally planned. In the theoretical model, A should be more efficient than B, but in practice the opposite can happen;
- ***Equity of cost (time access) to any memory location***. In real machines, RAM is not uniform from the point of view of access costs. There is a hierarchy of memory speed involving registers, various levels of cache, global, non-volatile, etc. Even within the same level of the hierarchy the more distant the memory module is from the processor, the longer it takes to access any data contained therein;
- ***Unlimited word size***. Both models (RAM and PRAM) assumes that the processors are able to manipulate words (instructions and data) of any size, without any restriction. This does not happen in practice and, when a big word has to be processed, it must first be "broken" in two or more before it can be manipulated, which takes several processing cycles, invalidating the fundamental assumption of unity cost of the models;
- ***Pipelining***. As described in Section 3.1, the *pipelining* available in all modern processor architectures allows the decoding of an instruction while another is running, greatly improving the performance of the programs, sequential or parallel. Again, the RAM and PRAM models do not consider this technique in their definitions;
- ***Synchronism***. The PRAM model requires strict synchronization in the execution of instructions. At each step of the program, all running processors must be at the

same instruction. However, various factors – such as different memory access times, the operating system in use and the fact that different instructions have different execution times, for example – may lead to an inefficient implementation of the program, because the faster processors must wait for the slower ones in order to synchronize and then execute the next instruction of the program;

- ***Distance between model and architecture***. Unlike what happened with the *von Neumann* architecture and the RAM model, which ended up becoming the *de facto* standard in the world of sequential computing, no model in parallel computing has established itself as dominant. Several proposals co-exist, each more suitable for a particular purpose, and the requirements of the PRAM model end up not being satisfactorily met by any of them.

Brent's Theorem

Brent's Theorem [26] states that, in PRAM model, any algorithm that requires time T_p with p processors can be simulated with q processors in time T_q, where $q \leq p$ and

$$\frac{W}{q} \leq T_q \leq \frac{W - T_p}{q} + T_p$$

The demonstration is simple. Let W be the amount of work done by the algorithm or, in other words, the product of time spent on its execution and the number of performed transactions. The work W can then be divided into

$$\sum_{i=1}^{T_p} W_i \text{ sub-works}$$

where W_i is the work actually performed during step i. If q processors are available in step i, then this task takes $\lfloor \frac{W_i}{q} \rfloor$ units of time to complete. However, since $W_i \leq q$, then $q - W_i$ processors are idle while the other W_i perform some work. So we can only say that, in general, step i takes $\lceil \frac{W_i}{q} \rceil$ units of time at most. Therefore, we have:

$$\sum_{i=1}^{T_p} \frac{W_i}{q} \leq T_q \leq \sum_{i=1}^{T_p} \lceil \frac{W_i}{q} \rceil$$

Since $\lceil \frac{W_i}{q} \rceil \leq 1 + \frac{W_i - 1}{q}$ for work W_i, we have that

$$\frac{W}{q} \leq T_q \leq \frac{W - T_p}{q} + T_p.$$

Concurrent Accesses

When developing algorithms on PRAM model, one must consider the existence (or not) of concurrent accesses to the same position of the shared memory, that is, policies to address conflicts of reading/writing should be established. Thus sub-models that describe how such conflicts should be managed were defined and are presented next [161, 195, 115]:

- ***EREW*** *(**E**xclusive **R**ead **E**xclusive **W**rite)* – this model does not allow read/write conflicts;
- ***CREW*** *(**C**oncurrent **R**ead **E**xclusive **W**rite)* – this is the standard model of PRAM. Several processors can read the same global memory location in the same execution step, but concurrent write operations are not allowed;
- ***CRCW*** *(**C**oncurrent **R**ead **C**oncurrent **W**rite)* – simultaneous read and write operations are allowed. Since this model leads to the possibility of access conflicts, several rules have been proposed to solve such conflicts. The most used are [110]:
  - ***Common*** – all processors making a write access to the same memory location must write the same value;
  - ***Arbitrary*** – among multiple processors trying to write in the same memory location, only one is randomly chosen;
  - ***Priority*** – among multiple processors trying to write in the same memory location, the one with the smallest index is chosen;
  - ***Combination*** – the data to be written follows a treatment rule. Among them, the maximal value can be chosen, adding them up, etc.

Extensions to PRAM Model

Given that PRAM is a purely theoretical model, not applicable in real machines, several extension proposals were suggested, trying to somehow eliminate or minimize its problematic points, described in section A.1. The following is a list of those proposals and a brief description of each one:

- ***APRAM*** *(**A**synchronous **P**arallel **R**andom **A**ccess **M**achine* [50]) – has been proposed to try to approach the PRAM model to real machines, since most of MIMD parallel computers available on the market (at that time) were potentially asynchronous. To achieve this goal in this model there is the introduction of explicit synchronization barriers. There is also a shared global memory which is used as a mean of asynchronous communication – if a processor needs to communicate with the other(s), it just writes the value in a memory location, without waiting for a reading step.

In the model, a parallel program is seen as a collection of *processes*. These, in turn, are each composed by a sequence of atomic operations[1], called *events*. There are three types of events: ***Read event***, which is the one that consults the shared memory, ***Write event*** is the one which writes (changes) the shared memory and a ***Local event*** is the one that carries out operations in local memory, either a read/write or a computation. It is also assumed that reading and writing events are able to access a block of memory in an atomic operation.

The precedence relation, defined as "run before" and represented by the symbol "$\rightarrow$" exists if, given two events s_1 and s_2 in the process P, if $s_1 \rightarrow s_2$, then s_1 runs before s_2. Or, if an event s_2 reads a variable x and s_1 is the event that wrote the data; or if there is an event s_3 for which $s_1 \rightarrow s_3$ and $s_3 \rightarrow s_2$. Two separate events, s and t are called *concurrents* if $s \nrightarrow t$ and $t \nrightarrow s$.

In turn, a *computation* is defined as a sequence S of events that satisfy the constraint that, given two events s_1 and s_2, if $s_1 \rightarrow s_2$, then s_1 appears before s_2 in S. This leads to the need of a read/write sequence in order to achieve synchronization.

If a process P_1 writes in a memory location and the same position is then overwritten by a process P_2 before any other process read such a position, the event P_1 does not impose any restrictions in the sequence of allowed events.

- ***H-PRAM*** – The ***H****ierarchical* ***PRAM*** [128] is presented in the article *"A Practical Hierarchical Model of Parallel Computation"* . One of the authors' motivations for the development of the H-PRAM is the difficulty often encountered when trying to conciliate two aspects when creating a model: ***simplicity*** and ***reflectivity***. According to the article, even though the *simplicity* is crucial so that the model can be useful in the real world, it must also allow efficient use of a realistic system of parallel computing. In other words, a good algorithm developed in the model should be able to be translated into a good algorithm in the target system. If this happens, it is said that the model is *reflective*, which is what H-PRAM offers.

  H-PRAM is unique because, unlike other PRAM extensions – which usually change fundamental characteristics of the original model, quite hindering the use of already developed algorithms – its proposal is the usage of PRAM as a sub-model, without modifying any feature of it, using a dynamically configurable PRAMs hierarchy which can synchronously communicate.

  Despite the fact that H-PRAM does not change any characteristic of the original

[1] An operation is defined as ***atomic*** if, during its code execution, a processor has exclusive access (read or write) to a memory location. This will prevent that any other hardware element (processor or I/O device) read or write from this memory location until the atomic operation is finished. It implies indivisibility and irreducibility, so any atomic operation must be performed entirely or, case it fails, all performed operations must be discarted. Therefore, if two or more processors try to perform atomic operations at the same memory location, this ultimately will serialize all the read/write operations.

PRAM, a new feature was added: a *partition function*, whose goals are to divide a task into smaller tasks, which are in turn solved by PRAMs with fewer processors, and to add a controlled form of asynchronism.
Once the partition function is called, it divides the problem into independent subsets and assigns a synchronous PRAM to each one. The splitting process can continue indefinitely, recursively, until the subtasks reach their threshold, where each subtask can be solved by only one processor. The Figure A.1 depicts this process [63].

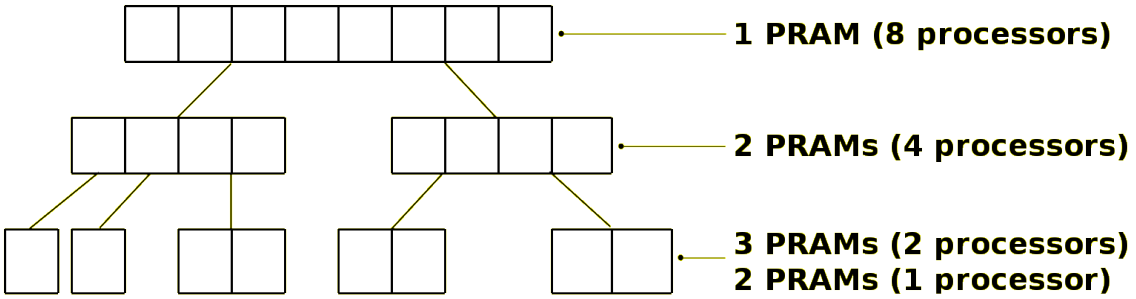


Figure A.1: *H-PRAM macro structure.*

Tasks being performed at a certain level operate asynchronously; however, the return to the previous level of the hierarchy is only possible after a synchronization between all tasks.
The H-PRAM model is divided in two sub-models, these differing in how they manage concurrent memory accesses:

- ***Privative H-PRAM*** – the partition function not only divides the tasks between processors but does the same with the system's shared memory, so that each sub-PRAM has its own private memory block, disjoint from the other private memories in the various sub-PRAMs sets;
- ***Shared H-PRAM*** – no partition is performed in the shared memory. Each sub-PRAM in the hierarchy has, therefore, access to all available memory.

Since one of the fundamental goals of the H-PRAM model is to be suitable to real machines, it admits for this two parameters: ***latency*** (delay in communication start between processors) and ***synchronization cost***. These parameters are defined according to the number of processors in communication and synchronization, respectively, and their values are specific to the target architecture.

- ***Phase PRAM***: In *"A More Practical PRAM Model"* [102] the author presents a new approach, abandoning the rigid synchronization scheme required by PRAM.
  Gibbons starts arguing that there are several difficulties when trying to convert PRAM algorithms for real MIMD machines and making several comments on some of them. The first refers to the fact that the PRAM model tacitly assume that each

processor can access any memory location in unit time, which is not true in real machines.

The second difficulty appears when it is observed that real MIMD machines are inherently asynchronous, allowing each processor to execute its own set of instructions, regardless of what others are doing. On the other hand, PRAM assumes that all processors execute the same instruction, in rigid steps and controlled by a single clock.

The author argues that his proposal to abandon the rigid timing imposed by PRAM leads to a better adaptation to real shared memory MIMD machines. As in the PRAM, the proposed asynchronous model consists of a set of $\boldsymbol{P}$ processors, each one with its own local memory and communicating through a global memory. However, unlike PRAM, each processor can perform a specific set of instructions and there is not a global clock.

The *Phase* PRAM extension proposes the adoption of stages, in which each processor works asynchronously and between each phase (computation), a synchronization instruction is used. This instruction, however, is not necessarily performed by all processors P of the system, but by a subset $\boldsymbol{S}$ such that $S \subseteq P$.

Formally, an instruction (or step) of synchronization in the subset S is a logical point of computation where each processor $p \in S$ waits for others in the sub-set before continuing with its local program.

Therefore, the program running on each processor consists of a series of instructions executed independently of the other processors, with the series separated by synchronizations. Prior to synchronization step, all instructions of all processors in S must be completed before any processor in S can start the next phase of computing.

In the *Phase* PRAM model, the author also provides a family of sub-models, differing in the types of synchronization, the shared memory access cost and way of read/write access to a memory location. They are:

- ***Phase*** **PRAM with subset synchronization** – here, multiple disjoint sets can synchronize independently and in parallel. The synchronization cost in subset S applies only to processors in S and is proportional to the set size;
- ***Phase*** **PRAM with all-processors synchronization** – multiple and parallel synchronizations are not allowed. That is, it is only possible to synchronize all processors involved in computation. Three options are available: the set S is equal to the set P; the set S is equal to the number of processors allocated to the program; and the set S corresponds to all active processors running the program;
- ***Phase*** **PRAM accounting for a communication latency**: the model allows

(but does not require) to take into account communication cost with shared memory. If the cost is considered, the delays are set with fixed size: a global reading takes time $2 * d$ and a global writing takes time d. If the value is not considered, then both read/write takes unit time;

- ***Phase* PRAM concurrent** – the model allows or does not allow concurrent readings and writings.

- ***QRQW PRAM***: The *Queue-Read Queue-Write* model proposes the inclusion of a new rule (named *queue*) for the managment of the concurrent accesses, in addition to those detailed in Section A.1, allowing that the cost from memory contention to be taken into account [102, 104, 105, 103, 63].
  Basically, the proposal of QRQW PRAM is that any memory location can be read or written by an undefined number of processors in each step of computing. However, concurrent access to a given memory location is allowed one at a time.
  It is defined as a synchronous model, with shared memory; however, each processor may have its own local memory. The entire communication process between processors is performed using shared memory. Being a synchronous model, between each computing step, a synchronization process must be performed. Figure A.2 illustrates this model. In essence, all execution in QRQW PRAM consists of a set of steps, each of which is divided into the following parts:

  - ***Reading***: r_i cells in shared memory are accessed by processor $\boldsymbol{i}$;
  - ***Calculation***: c_i calculations are locally performed by processor $\boldsymbol{i}$;
  - ***Writing***: w_i values are written by each processor $\boldsymbol{i}$, in w_i positions of the shared memory.

  Even considering the costs of concurrent memory access, the QRQW PRAM completely abstracts the implementation aspects in real machines, such as latency access to shared memory, synchronization cost between processors, non-uniform time access to data, etc. Nevertheless, the authors show [105] that the model can be easily emulated in the BSP model.
- ***BSR*** (**B**roadcasting with **S**elective **R**eduction) – First proposed by Akl and Guenther [3], it consists of the creation of a new rule to handle memory accesses, in addition to those defined in Section A.1: the *BROADCAST* instruction [45].
  Such instruction allows all the $\boldsymbol{N}$ processors to write to all $\boldsymbol{M}$ memory locations at the same time, which is done through the following steps (or phases):

  - ***Broadcast*** – in this phase, all the $\boldsymbol{N}$ processors write in all $\boldsymbol{M}$ memory positions. The processor P_i, $1 < i < N$ package a record containing two fields: a tag g_i and a set of data d_i; the tag g_i will identify the memory locations in which d_i should be stored;

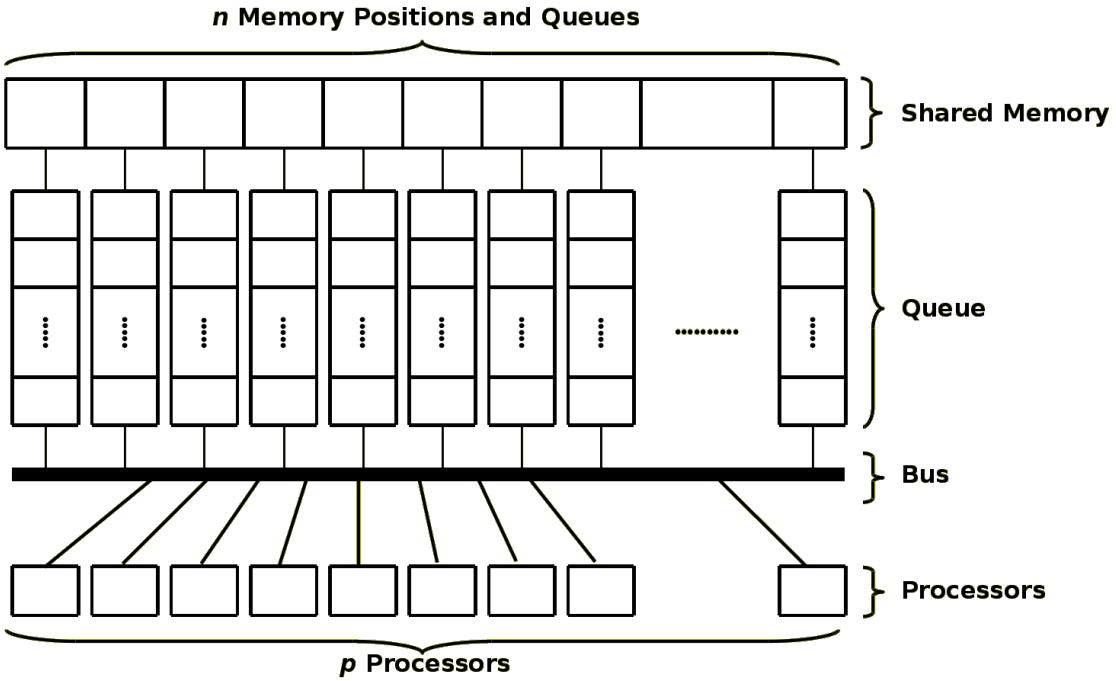


Figure A.2: *QRQW-PRAM Macro Structure.*

- ***Selection*** – here, after the data is received in each memory location U_j, $1 \leq j \leq M$ a *switch* associated with this memory location will select a subset of d_i, comparing the value of g_i with a threshold value l_j, using a selection rule σ;
- ***Reduction*** – Finally, the data received is *reduced* to a single value using an associative binary reduction $\Re$.

Each one of the three phases is carried out simultaneously by N processors and M *switches* in M memory locations. The Figure A.3 depicts the process.
The selection rule, σ, can be any one among the following relational operators:

$$\{<, \leq, =, \geq, >, \neq\}$$

The reduction rule $\Re$ can be chosen from any of the following associative operators:

$$\{\sum, \prod, \wedge, \vee, \oplus, \cap, \cup\}$$

and correspond, respectively, to *sum*, *product*, *logical* ***and***, *logical* ***or***, *exclusive* ***or***, *maximum* and *minimum*.

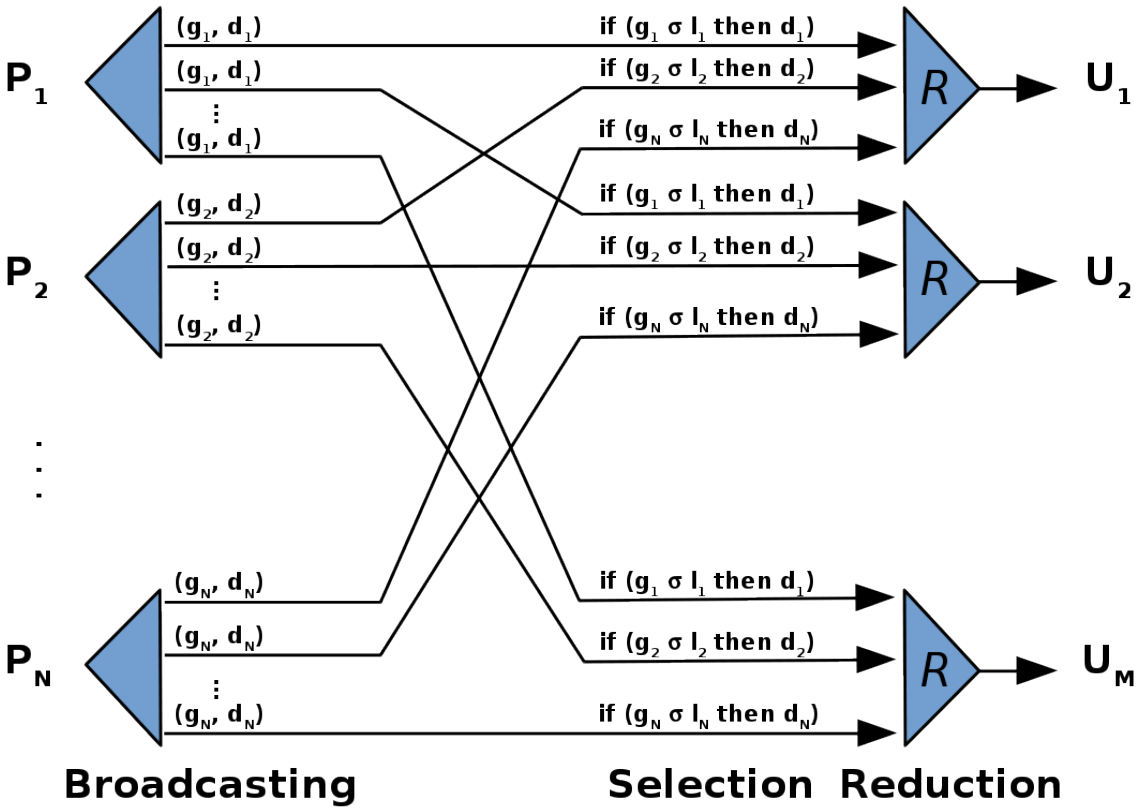


Figure A.3: *BROADCAST of an instruction in three phases.*