Dinâmica de Redes Booleanas Limiarizadas utilizando programação em GPU

William Lira Ferreira

Dissertação a ser Apresentada
ao
Instituto de Matemática e Estatística
da
Universidade de São Paulo
para
obtenção do título
de
Mestre em Ciências

Programa: Ciência da Computação
Orientador: Prof. Dr. Ronaldo Fumio Hashimoto

São Paulo, Agosto de 2013

Dinâmica de Redes Booleanas Limiarizadas utilizando programação em GPU

Esta dissertação trata-se da versão original
do aluno William Lira Ferreira.

Agradecimentos

A Deus, pela saúde, força de vontade e determinação, que estão sempre presentes em minha vida.

A minha mãe Francisca, irmãos Queli e Kleber e toda minha família que com carinho e esforço não mediram esforços para que eu chegasse até mais esta etapa de minha vida. A minha avó, Josefa, que infelizmente deixou este mundo, mas terá sempre sua alegria, bom humor e fé lembrados. Agradeço a Elienaide por ser esta pessoa especial que sempre esteve ao meu lado, com paciência e apoio, mesmo nas horas mais difíceis.

Ao professor Ronaldo pela paciência, dedicação e incentivo na orientação, possibilitando a conclusão desta dissertação, também pelas diversas horas de conversa e discussão, na busca de uma solução para o desafio enfrentado.

Ao professor Gubi pela prontidão e apoio na disponibilização de um ambiente para realização dos experimentos e principalmente pelo tempo de processamento das máquinas e GPUs sobre sua responsabilidade. Neste mesmo sentido, também agradeço aos professores David Martins e Luiz Rozante e o pessoal da UFABC pelo compartilhamento de conhecimento e acesso à alguma de suas máquinas e GPUs.

Aos amigos Paulo, Sylvio e Luiz Felipe pelas horas de estudo em conjunto para atingir bons aproveitamentos nas disciplinas do curso, e, em especial, aos amigos Tales e Sérgio pelas longas conversas e discussões produtivas que contribuíram para a evolução deste trabalho.

Por fim, ao IME e a USP, pela infraestrutura e oportunidade que foi me dada para alcançar este objetivo acadêmico/profissional, que é o Mestrado em Ciências.

À todos, muito obrigado!

Resumo

A área de Biologia Sistêmica procura estudar os vários mecanismos celulares e seus componentes, os quais realizam diversas interações e reações químicas. Nessa área, para representar estas interações e reações no nível gênico, isto é, o relacionamento entre os genes e sua expressão gênica, é comum o uso de modelos matemáticos e computacionais de Redes de Regulação Gênica (GRN - do inglês *Gene Regulatory Network*). Existem diversos modelos para representar GRNs, os quais possuem relação temporal síncrona ou assíncrona e caracterizados como determinísticos ou probabilísticos. Um dos modelos de grande interesse e estudado na literatura é o de Redes Booleanas (BN - do inglês *Boolean Networks*), no qual os genes podem assumir dois estados, ativo ou inativo - se expressos ou não, respectivamente, e, além disso, possuir uma função booleana associada. A evolução dos estados dos genes no tempo, é dada pela transição de um estado da rede para outro e assim sucessivamente. Esta evolução forma uma dinâmica temporal, na qual as transições de estado são ditadas pelas funções booleanas associadas a cada gene. Como o número de estados é finito, em algum momento a rede deve voltar para um estado no qual já esteve anteriormente, formando um ciclo. Este ciclo é chamado de atrator.

Na análise da dinâmica de uma BN a identificação dos atratores possui grande importância visto a interpretação biológica dos mesmos. Outra informação relevante é sobre o tamanho da bacia de atração de cada atrator. A bacia de atração é formada pelo conjunto de todos os estados, que através de alguma transição alcançam o atrator. Neste trabalho foi realizada a análise da dinâmica, contemplando a identificação dos atratores e contagem do tamanho das bacias de atração para um caso particular de BN, conhecido como Redes Booleanas Limiarizadas, onde apenas uma classe de funções booleanas está disponível.

A análise da dinâmica de uma BN é um desafio computacional devido ao elevado número de elementos integrantes da simulação da rede. Para amenizar este problema, propõe-se o estudo de métodos de alto desempenho e também a implementação de algoritmos em GPUs (*Graphics Processing Unit*) utilizando CUDA (*Compute Unified Device Architecture*). Com isso, espera-se um aumento no número de genes das redes processadas e também melhoria no desempenho com relação a computação utilizando somente processadores (CPUs) convencionais. Como produto deste trabalho é esperado um software, que será integrado à uma ferramenta de inferência de redes booleanas existente como módulo de análise da dinâmica de redes.

Palavras-chave: Redes Booleanas, Atratores, Bacias de Atração, Programação em GPU, CUDA

Abstract

Systems Biology area aims to study various cellular mechanisms and its components, which perform various interactions and chemical reactions. In this area, to represent these interactions and reactions at the gene level, ie, the relationship between genes and their gene expression, is common to use mathematical and computer models of Gene Regulatory Networks (GRN). There are several models to represent GRNs, which have temporal relationship synchronous or asynchronous and characterized as deterministic or probabilistic. A model of great interest and attention in literature is Boolean Network model (BN), in which genes can assume two states, active or inactive - if are expressed or not, and moreover, owning a boolean function associated. The evolution of all genes in time is given by a transition from a network state to another. This evolution forms a temporal dynamics, in which state transitions are controlled by boolean functions associated with each gene. As the number of states is finite, at some point, the network must return to a state already visited previously, forming a cycle. This cycle is called attractor.

The analysis of network dynamics on BNs to identify attractors has great importance, and is motivated by your biological interpretation. Other relevant information is about the size of basin of attraction of each attractor. The basin of attraction is composed by all states, that through some transition steps, reaches the attractor. In this work was performed dynamic analysis, covering the identification of attractors and count the size of the basins of attraction for a particular case of BN, known as Restricted Boolean Networks, where only one class of Boolean functions is available.

Due to the high number of elements of network simulation, the analysis of the dynamics of a BN is a computational challenge. To ease this problem, we propose the study of methods of high performance computing and also the implementation of algorithms in Graphics Processing Unit (GPU) using Compute Unified Device Architecture (CUDA). Thus, we expect an increase in number of genes processed and also improvement in performance with respect to computation using only conventional processors (CPU). As a result of this work is expected a software to be integrated into a tool of Boolean Network Inference as a module of dynamic networks analysis.

Keywords: Boolean Networks, Attractors, Basin of Attraction, GPU Programming, CUDA

Sumário

Lista de Figuras

Lista de Tabelas

Capítulo 1

Introdução

Sistemas naturais e artificiais são estudados em diversas áreas das Ciências. Em muitos casos, dada a sua complexidade, para o entendimento dos mesmos são utilizadas simulações numéricas e modelos computacionais, pois o tratamento analítico torna-se inviável. Na modelagem computacional são realizadas simplificações para representação dos fenômenos de interesse. As simplificações são possíveis pois estes fenômenos permitem ser tratados de modo discreto e não contínuo, obviamente até certo nível de abstração. Por exemplo, no contexto de um "sistema social", podemos nos alimentar ou não, estar satisfeitos com relação a determinado estímulo ou não, nos comunicar ou não. Já contexto de um sistema biológico, podemos tratar um gene como expresso ou não, determinar a ocorrência de uma reação química ou não, observar a existência de certa proteína ou não, etc. Tratando essas escolhas como decisões binárias, estes sistemas são modelados por lógica computacional, utilizando relações de causa e efeito, as quais podem ser descritas através de uma rede. Esta rede é então simulada, tornando possível algum entendimento sobre sistema em questão. Para a idealização computacional de sistemas biológicos foi cunhado o termo "Bioinformática" (Hogeweg [2011]), definido originalmente como *"the study of informatic processes in biotic systems"* (P. Hogeweg [1978], Hogeweg [1978]). Outros autores definem Bioinformática como: *"Bioinformática está relacionada ao rápido desenvolvimento da área de Ciência da Computação devotada a armazenar, organizar e analisar sequências de DNA e proteínas"* (Lodish *et al.* [2000]); *"Bioinformática é o uso de métodos computacionais no estudo de genomas"* (Brown [2002]).

A Bioinformática tem como base o *Dogma Central da Biologia Molecular*, o qual estabelece que o estado funcional de um organismo celular é amplamente determinado pela sua expressão gênica (D'Haeseleer *et al.* [1999]). Partindo deste pressuposto, são enunciados diversos problemas, os quais são modelados utilizando pensamento sistêmico e com isso os organismos celulares e seus componentes são tratados de forma modular e sistemática. Neste cenário, os genes, as proteínas, fatores de transcrição, e as demais interações entre estes elementos formam o que é chamado de Rede de Regulação Gênica (GRN - *Gene Regulatory Network*). Uma GRN é representada por um diagrama onde os nós representam os genes, proteínas ou fatores de transcrição; as arestas entre pares de nós indicam que existe uma interação entre os mesmos. A fim de estudar e concretizar a abstração das GRNs, são utilizados modelos computacionais contínuos ou discretos, usados de forma determinística ou estocástica. Estes modelos promovem a caracterização do sistema em estudo, permitindo análises em diversos níveis de informação. O modelo discreto mais simples é o de Rede Booleana (BN - *Boolean Network*), introduzido por Kauffman (Kauffman [1969] e Kauffman [1993]), o qual é um dos mais utilizados no estudo das propriedades dinâmicas de GRNs. Neste modelo, cada gene pode estar ativo ou não, assumindo valores 0 ou 1 respectivamente. O estado do sistema é determinado pelo valor assumido por cada gene em um dado instante de tempo.

Entre outros modelos, podemos destacar ainda o modelo de Rede Booleana Probabilística (PBN - *Probabilistic Boolean Network*) proposto por Shmulevich *et al.* [2002]. Friedman *et al.* [2000] introduziu o modelo de *Rede Bayesiana* na identificação de dados regulatórios para reprodução de

relacionamentos conhecidos entre genes. Entre os modelos contínuos podemos citar os modelos de equações diferenciais propostos por Goodwin [1963] e Wahde e Hertz [2000]. Para maiores informações sobre os diversos modelos e um estudo comparativo, veja os trabalhos de de Jong *et al.* [2003], Karlebach e Shamir [2008], Akutsu *et al.* [2008]

Neste trabalho, focamos no modelo de Rede Booleana. Este modelo, apesar de sua simplicidade, é capaz de representar, através de sua dinâmica, diversos fenômenos biologicamente significativos, descrevendo qualitativamente o comportamento global de uma GRN. É usado um caso particular de BN chamado de *Rede Booleana Limiarizada* (Li *et al.* [2004], Higa *et al.* [2010]), onde os estados continuam sendo tratados de forma discreta e determinística porém, somente uma classe de funções booleanas é utilizada para representar as relações entre genes. Este modelo foi utilizado com êxito na modelagem do ciclo celular da levedura (Li *et al.* [2004]) e também em algoritmos de inferência de redes a partir de dados temporais (Higa *et al.* [2010], Andrade [2012] e Higa *et al.* [2011]).

A dinâmica de uma BN é definida através da transição de um estado da rede para outro, através da aplicação das respectivas funções booleanas, assim sucessivamente. Pela característica determinística (cada estado possui um único sucessor), sendo o número de estados finito, após certo número de transições, ocorrerá a repetição de um conjunto de estados, formando um ciclo. A sequência de estados que compreende este ciclo chama-se *atrator* e o conjunto de estados que alcançam o mesmo atrator formam uma *bacia de atração*. Estas informações podem ser utilizadas na determinação da estabilidade de uma rede. A estabilidade é a maneira como a rede se comporta diante de possíveis perturbações do ambiente externo.

Acredita-se que GRNs reais são altamente estáveis na presença de perturbações ocasionadas por fatores externos, seja sobre um gene isoladamente ou vários. No modelo de rede booleana, isto representa que, quando um número mínimo de genes são perturbados, estes genes mudam de valores, porém o novo estado global da rede continua na mesma bacia de atração, acabando por alcançar, através de uma trajetória, o mesmo atrator anterior à perturbação (Shmulevich *et al.* [2005], p252; Kauffman [1993], p190). Neste sentido, atratores com grandes bacias de atração conferem uma alta estabilidade para o sistema. O reflexo da estabilidade na rede regulatória é que as células tendem a manter seu estado funcional normal, mesmo quando submetidas a perturbações externas. Ainda como fator motivacional da importância dos atratores, o trabalho de Kauffman (Kauffman [1993]) interpreta os atratores de uma BN como diferentes tipos celulares, argumentando que diferentes células são caracterizadas por seu padrão recorrente de expressão gênica, de certa forma correspondente aos atratores de uma BN.

Para redes booleanas com maior número de genes, a identificação dos atratores e contagem do tamanho das bacias de atração torna-se problemática e intratável, pois o crescimento do espaço de estados é exponencial, sendo que para uma rede com n genes temos 2^n estados. Diversos trabalhos tratam do assunto, como Wuensche [1998], Irons [2006], Skodawessely e Klemm [2011], Dubrova e Teslenko [2011], Müssel *et al.* [2010], entre outros. As soluções são computacionalmente intensivas, face o crescimento exponencial do número de estados do sistema com o aumento do número de genes. A modelagem utilizado redes booleanas limiarizadas, permite a representação das funções booleanas em forma matricial. Neste aspecto, com objetivo de amenizar o problema e permitir o processamento de redes com maior número de genes, neste trabalho propõe-se o uso de tecnologias de alto desempenho e programação utilizando GPU (*Graphics Processing Unit*), para identificação dos atratores e contagem do tamanho das bacias de atração. O uso de programação em GPU vem ganhando cada vez mais espaço no desenvolvimento de aplicações científicas e também na indústria, devido ao alto número de núcleos destes dispositivos e também pela sua capacidade de processamento paralelo. A implementação dos algoritmos foi realizada utilizando

o modelo de programação e arquitetura CUDA™, desenvolvido pela NVIDIA © [1].

Foram propostos dois algoritmos para GPU. O primeiro é baseado na estrutura de grafos e em um algoritmo de componentes conexas para GPU (Hawick *et al.* [2010]). Este algoritmo mostrou bons resultados e tempos de execução, porém possui limitação quanto à quantidade de memória disponível, visto que o espaço de estados, que é carregado na estrutura de grafo, facilmente ultrapassa o limite deste recurso. Já o segundo algoritmo, realiza *trajetórias em paralelo* e não considera nenhuma estrutura de dados a priori, desta forma, o processamento se dá pela paralelização das trajetórias, obtendo-se a identificação do atrator de cada estado da rede. Como esta abordagem não armazena o espaço de estados em memória, este recurso não se torna um fator limitante. A implementação do segundo algoritmo está preparada para executar em múltiplas GPUs, permitindo maior nível de escalabilidade.

A distribuição do software desenvolvido foi feita através de uma biblioteca de software (C/C++), um aplicativo de linha de comandos e para uso com experimentos gerais, um aplicativo que permite sua execução pela web. Ao final foi realizado um comparativo do tempo de execução entre a implementação utilizando GPU, a implementação utilizando método convencional e uma implementação utilizando método de resolução de problemas de *Satisfatibilidade Booleana* (SAT).

1.1 Objetivos

Durante o desenvolvimento deste trabalho estudamos métodos existentes para identificação dos atratores e contagem do tamanho das bacias de atração. Sabemos que o problema é de difícil resolução, face o crescimento exponencial do espaço de estados, com aumento do tamanho das redes a serem analisadas. Dada a importância dos atratores e sua consequente aplicação no contexto de GRNs, este trabalho tem como o objetivo o desenvolvimento de métodos de alto desempenho, para identificação dos atratores e contagem do tamanho das bacias de atração de uma BN, possibilitando assim a manipulação de redes com maior número de genes. Desta forma, é esperada também a diminuição no tempo de execução das instâncias de problemas atuais, realizando a manipulação completa da rede, a qual se dá para redes com até 29 genes no máximo (por exemplo, utilizando as ferramentas: *BoolNet* (Müssel *et al.* [2010]), *GINSim* (Fauré *et al.* [2006]), ou a implementação serial padrão de referência - veja Seção 2.1.4). Os métodos de alto desempenho utilizados são de programação paralela utilizando programação para GPUs, sobre a plataforma CUDA.

Utilizamos o modelo de rede booleana limiarizada, por este ser alvo de modelagem de GRNs reais e principalmente pela sua representação matricial. Esta última característica é interessante, visto que na programação para GPU operações sobre matrizes são comuns e relativamente simples de serem feitas. A programação em GPU possui certas limitações quanto às estruturas de dados utilizadas e a representação matricial, mesmo sendo simples, parece ser suficiente para permitir a simulação da rede, oferecendo diversos graus de paralelismo possíveis, inclusive com uso de múltiplas GPUs.

Como objetivo secundário, espera-se a realização de um comparativo do tempo de execução entre os métodos existentes e a solução proposta. Este comparativo validará o algoritmo proposto e também possibilitará a construção de curvas dos tempos de execução. As instâncias geradas para o comparativo são de redes geradas de forma aleatória.

[1] www.nvidia.com

1.2 Contribuições

Este trabalho contribui na direção dos diversos estudos já realizados sobre redes booleanas, com foco em desempenho, tornará possível a manipulação e simulação de redes que antes eram simuladas apenas parcialmente. Por ser distribuído em forma de biblioteca, o software permite ser embarcado diretamente em outras aplicações, além disso, pode ser executado através da web, utilizando uma instalação de servidor que conta com uma ou mais GPUs NVIDIA, permitindo ser executado para o usuário final sem restrições de ambiente.

Dado o problema de identificação dos atratores e contagem do tamanho das bacias de atração, as principais contribuições deste trabalho são:

- Pesquisa e levantamento de algoritmos de alto desempenho existentes para solução do problema;
- Proposta de novos algoritmos paralelos para serem implementados utilizando programação para GPU;
- Implementação dos novos algoritmos e disponibilização de biblioteca de software, aplicativo de linha de comandos e aplicação web;
- Possibilitar a solução do problema alvo em menor tempo;
- Possibilitar a análise de redes booleanas com maior número de genes.

1.3 Organização do Trabalho

Os próximos capítulos deste trabalho estão organizados da seguinte forma:

Capítulo 2 - Revisão Teórica: Será realizada uma revisão teórica dos conceitos relacionados a redes booleanas e também de programação para GPU. Primeiro serão abordados conceitos relativos às GRNs e apresentados formalmente os modelos de rede booleana e de rede booleana limiarizada, incluindo discussão sobre seu uso. Outros conceitos teóricos interessantes apresentados neste capítulo incluem: análise da dinâmica, estabilidade da rede, algoritmos básicos, ferramentas existentes, tipos de redes e redes aleatórias. Por um lado técnico e ferramental, serão abordados aspectos da programação paralela e a programação para GPU, detalhando-se a estrutura da arquitetura CUDA e como se dá seu funcionamento.

Capítulo 3 - Projeto: O problema a ser resolvido é apresentado e as alternativas de paralelização do ponto de vista de programação para GPU e CUDA são levantadas e discutidas. A metodologia utilizada na pesquisa e desenvolvimento da solução é brevemente apresentada. O software, incluindo o desenho da arquitetura base e os algoritmos implementados são mostrados em detalhe.

Capítulo 4 - Resultados: Neste capítulo serão mostrados alguns resultados da implementação. Além disso, é mostrado o comparativo de tempos de execução entre as implementações de referência escolhidas.

Capítulo 5 - Considerações Finais: São dadas as considerações finais e conclusões deste trabalho. Por fim, daremos algumas perspectivas de trabalhos futuros.

Capítulo 2

Revisão Teórica

Neste capítulo, serão introduzidos conceitos de redes booleanas, detalhando-se o problema de *análise da dinâmica*, um problema importante e de difícil resolução, pela entrada crescer exponencialmente em relação ao número de genes analisados. A solução deste problema provê informações tais como identificação dos atratores e tamanho das bacias de atração.

Também serão apresentados conceitos de programação paralela e programação para GPU. Será detalhada a arquitetura e modelo de programação CUDA, bem como esta pode ser utilizada para prover poder computacional adicional na solução do problema alvo deste trabalho.

2.1 Fundamentos

Modelos matemáticos e computacionais de redes de regulação gênica têm sido desenvolvidos para auxiliar o estudo das interações entre genes ou proteínas de sistemas biológicos. Estes modelos são representações formais simplificas da realidade. Cada modelo possui suas características e pode ser útil em ajudar a responder diferentes perguntas. A importância de utilizar modelos é que podemos adquirir novos conhecimentos sobre determinado fenômeno. Embora fenômenos biológicos se manifestem de maneira contínua, é muito comum descrevê-los utilizando lógica binária (Shmulevich *et al.* [2002]).

Redes de Regulação Gênica (GRN) podem ser úteis no estudo de fenômenos biológicos (como ciclo celular) e doenças (como câncer), desta forma, o trabalho com tais redes, a fim de elucidar seu comportamento e como este descreve o fenômeno biológico em questão, é de grande importância e interessante objeto de estudo. Existem vários modelos de GRN, tanto discretos como contínuos. O modelo mais simples foi introduzido por Kauffman [1969] e é conhecido como Rede Booleana (BN - *Boolean Network*). Em seguida, este modelo foi modificado para expressar incerteza dando origem a Rede Booleana Probabilística (Shmulevich *et al.* [2002] e Zhang *et al.* [2007]). Friedman introduziu Rede *Bayesiana* como uma ferramenta para identificação de dados regulatórios e mostrou que a mesma pode reproduzir algumas relações biológicas conhecidas (Friedman *et al.* [2000]). Entre os modelos contínuos, podemos citar o modelo de equações diferenciais de Goodwin [1963].

Este trabalho tem foco no modelo de Rede Booleana. É utilizada uma variação deste modelo, chamada de *Rede Booleana Limiarizada*, descrita por Li *et al.* [2004] e utilizada nos trabalhos de Higa *et al.* [2011] e Andrade [2012].

2.1.1 Redes de Regulação Gênica

Uma Rede de Regulação Gênica é uma rede onde os nós representam genes ou proteínas e as arestas, entre pares de genes, indicam que existe uma interação entre estes, ou seja, o produto de um gene, afeta o produto do outro, fazendo existir uma relação de dependência entre os mesmos.

Estas arestas podem ser direcionadas, indicando ativação ou inibição. A estrutura de uma GRN é uma abstração das relações químicas entre os componentes do sistema biológico em estudo. A capacidade de inferir GRNs tornou-se possível graças ao desenvolvimento e avanço de tecnologias *high-throughput* permitindo aos cientistas realizar análises nos níveis de DNA e RNA. O tipo mais comum de dados providos por estas tecnologias são dados de expressões gênicas chamados *micro-array*, descritos por Schena *et al.* [1995] e Shalon *et al.* [1996]. O termo "expressão gênica" refere-se ao *transcriptoma* de um organismo. O transcriptoma é o processo pelo qual a informação codificada por um gene é utilizada para a síntese do produto gênico final, que geralmente é uma proteína (em alguns casos, o produto final pode ser um RNA funcional). Este processo é realizado por todas as formas de vida conhecidas - procariotos e eucariotos - para que os organismos mantenham-se vivos (Hache *et al.* [2009]). Desta forma, ele é o reflexo direto da expressão dos genes.

A abstração da Figura 2.1 ilustra o processo de transcrição, ou ciclo celular, o qual possui 3 níveis específicos (replicação do DNA, transcrição e tradução), que constituem o dogma central da biologia molecular. No nível do DNA, através da transcrição, ocorre produção de RNA mensageiro, também conhecido como mRNA. O mRNA, através da tradução, no nível de transcrição, produz então proteínas. Essas proteínas formam complexos que interagem entre si, integram sinais extracelulares e atuam na regulação das vias metabólicas recebendo e enviando sinais de realimentação. Assim o nível de expressão de cada gene depende tanto do valor de sua própria expressão quanto dos valores da expressão de outros genes em instantes de tempo passados, além de estímulos externos.

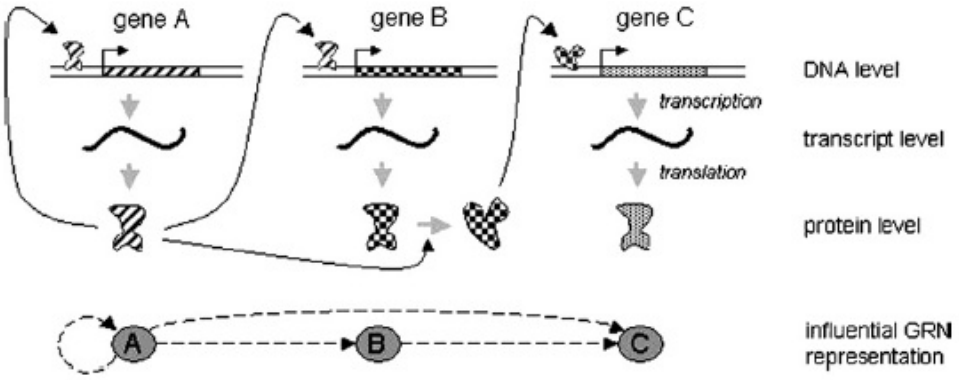


Figura 2.1: *Processo de transcrição e representação esquemática da rede de regulação gênica. Podemos notar que no nível do DNA, o gene **A** é auto-regulado pelo produto de sua própria expressão e que este regula o gene **B**. O gene **A** também exerce influência reguladora, através do gene **B**, diretamente no nível de proteína ou indiretamente no nível de DNA, ao gene **C**. No nível de GRN, a abstração da dinâmica química é realizada para realçar a influência entre os genes (Hecker et al. [2008]).*

Um desafio sempre presente em qualquer modelo de GRN é sobre sua utilidade. A fim de se obter o entendimento dos complexos padrões apresentados pelo comportamento dos genes e a identificação de suas interações, ferramentas como grafos são amplamente utilizadas. Com o uso de grafos, esta modelagem permite capturar o comportamento coletivo dos genes e o relacionamento entre eles de forma direta. Sua topologia exibe estes relacionamentos de forma explícita, os quais se mantêm independente da atribuição de algum valor numérico aos genes ou suas ligações. Por essa razão, os grafos são amplamente utilizados para denotar uma relação de dependência entre os genes (Pearl [1988]). Na parte inferior da Figura 2.1, temos a representação em forma de grafo da GRN apresentada. O grafo abstrai o relacionamento entre os genes e mostra apenas a relação de influência que cada gene exerce sobre outros genes, de forma direta ou indireta.

Pela dependência e a evolução temporal as GRNs permitem ser tratadas como sistemas dinâmicos. Vários modelos de GRNs foram propostos e podemos dividi-los em determinísticos e

estocásticos, podendo ser discretos ou contínuos. Em modelos determinísticos o estado da expressão gênica (nível de expressão de todos os genes considerados) em um dado instante de tempo e as interações regulatórias entre eles determinam, sem ambiguidade, o estado de expressão gênica no próximo instante de tempo. Nestes modelos há somente uma transição possível de um estado de expressão gênica para outro. Já em modelos estocásticos, um estado de expressão gênica pode gerar mais de uma transição no próximo instante de tempo. No caso dos modelos contínuos podemos citar modelos de equações diferenciais não lineares (Wahde e Hertz [2000]). Para o modelo discreto, temos que cada gene é representado por uma variável que armazena o valor dado pela expressão do gene. Todas as variáveis em conjunto formam um estado do sistema, o qual representa o sistema em um determinado instante de tempo. Cada variável está associada a uma função de transição, que define a transição para o próximo estado, modelando a ação conjunta do efeito da regulação sobre os genes. Neste trabalho, estamos interessados no estudo de uma variação do modelo de Redes Booleanas, o qual será detalhado em seguida.

2.1.2 Redes Booleanas

Redes Booleanas (*Boolean Networks*) foram introduzidas por Kauffman [1969]. Em termos gerais, em uma rede booleana, um gene é descrito como expresso ou não em um dado instante de tempo, assumindo os valores ativo (1) ou inativo (0). O valor de cada gene em um dado instante de tempo é definido através de uma função booleana específica, considerando o valor dos genes no instante de tempo anterior. O vetor de zeros e uns que descreve o nível de todos os genes é chamado de estado do sistema, ou estado global. Assume-se que as mudanças de estado são síncronas, tais como a cada passo de tempo, o nível de cada gene é determinado de acordo com os níveis de regulação no passo de tempo anterior e a aplicação da função de regulação. As redes booleanas podem ser descritas por um grafo direcionado, onde as arestas representam as interações entre os genes. O produto destas interações é representado por funções booleanas constituídas de operações básicas como *AND* ($\wedge$), *OR* ($\vee$), *NOT* ($\neg$).

Formalmente, segundo Kauffman [1969], uma rede booleana $G(V, F)$ é definida por um conjunto de vértices $V = \{v_1, v_2, ..., v_n\}$, os quais representam um conjunto de n genes, e um conjunto de funções booleanas $F = \{f_1, f_2, ..., f_n\}$, $f_i : \{0,1\}^{k_i} \rightarrow \{0,1\}$, uma para cada um dos genes, também conhecidas como funções de transição booleanas. Cada gene $v_i \in \{0,1\}, i = 1, 2, ..., n$ representa uma variável booleana. Se $v_i = 0$, dizemos que o gene v_i está inibido ou desligado. Da mesma forma se $v_i = 1$, dizemos que v_i está expresso ou ligado. O valor de v_i no instante de tempo $t + 1$ é determinado por uma das funções booleanas f_i e pelo valor dos k_i genes que possuem arestas incidentes em v_i, chamados de genes *preditores*, denotados por $v_{1i}(t), v_{2i}(t), \ldots, v_{ki}(t)$. Desta forma, existem k_i vértices adjacentes associados a v_i, também chamado de *gene alvo*. Na rede booleana, as funções booleanas são utilizadas na atualização dos genes alvo, considerando iterações discretas no tempo, sendo todos os genes atualizados de forma síncrona de acordo com a função booleana relacionada ao mesmo e os valores de seus respectivos genes preditores. Assim, valor de cada gene no próximo instante de tempo $(t + 1)$ é definido como:

$$v_i(t + 1) = f_i(v_{1i}(t), v_{2i}(t), \ldots, v_{ki}(t)) \tag{2.1}$$

Um *estado global da rede*, ou apenas *estado da rede* é um vetor binário, definido pelos valores de todos os genes em um dado instante de tempo. Isto é:

$$s(t) = (v_1(t), v_2(t), \ldots, v_n(t)), v_i(t) \in \{0,1\} \forall i = 1, 2, ..., n \tag{2.2}$$

O número de estados possíveis é 2^n, denotados por $s_0, s_1, \ldots, s_{2^n-1}$. Dado um estado no tempo t, a aplicação das funções booleanas de maneira síncrona leva a rede para o próximo estado no

tempo $t + 1$ de maneira determinística, pois há uma única função booleana associada a cada gene. Podemos representar estas transições de estado utilizando uma tabela ou grafo, chamado de *grafo de transições de estado*. Na representação por tabela, cada linha armazena o estado e a respectiva transição na rede, ou seja o próximo estado. Já no grafo de transições de estado cada vértice representa um estado da rede e uma aresta direcionada do estado s_i para s_j indica que as funções em F resultam na transição $s_i \rightarrow s_j$. Pelo número finito de estados e o comportamento determinístico, alguns dos estados serão visitados ciclicamente. Estes estados formam o que é conhecido como *atrator*. Os estados fora do atrator são chamados de *estados transientes*. O conjunto formado pela união dos estados do atrator com os estados transientes forma a *bacia de atração* (Kauffman [1969], Higa *et al.* [2011]). Os atratores são sempre cíclicos e podem ser formados por um ou mais estados. Uma transição sequencial de um estado em direção ao atrator é chamada de *trajetória*. Desta forma o grafo de transições de estado representa a *dinâmica da rede*, isto é todas as suas transições e trajetórias, contendo os atratores e bacias de atração.

A Figura 2.2 ilustra uma rede booleana, contendo a representação da rede booleana em forma de grafo, as funções booleanas, a tabela de transições de estado e um diagrama de transições de estado, sendo que estes dois últimos representam a dinâmica da rede. Em 2.2(a) as arestas direcionadas representam uma ativação, enquanto as arestas achatadas representam uma inibição, desta forma v_1 inibe v_2 e v_2 ativa v_1. O gene v_3 é ativado se v_1 ou (OR) v_2 estiverem ativados. Em 2.2(b) o relacionamento entre os genes é representado de forma lógica textual. Já 2.2(c) exibe em uma tabela todas as possibilidades de transições de estado possíveis para uma entrada de tempo t e saída em $t + 1$. Finalmente em 2.2(d) temos estas transições representadas em um diagrama de transições de estado, contendo a dinâmica do sistema (que da mesma forma pode ser vista na representação tabular). Por exemplo, se a rede estiver no estado 101 - genes ($v_1 = 1, v_2 = 0, v_3 = 1$), então o próximo estado desta rede será 001 - genes ($v_1 = 0, v_2 = 0, v_3 = 1$). Os estados 001, 010, 101 e 111 são estados do atrator.

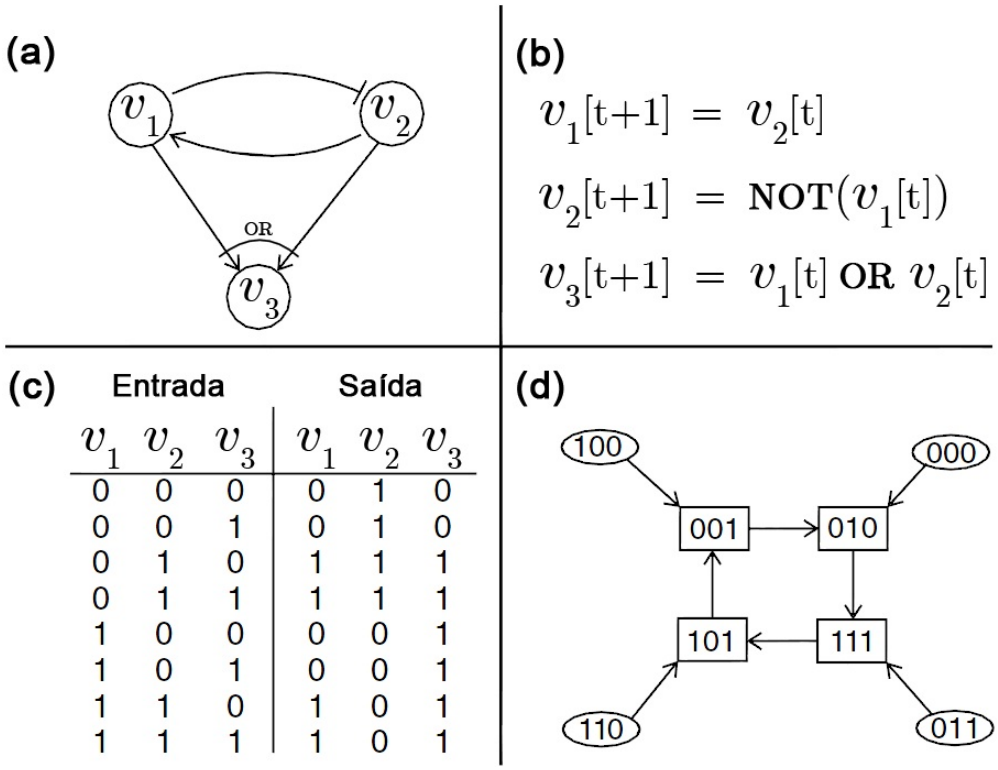


Figura 2.2: *Exemplo de uma rede booleana e suas representações. (a) Representação em forma de grafo. (b) Funções Booleanas. (c) Tabela de transições de estados. (d) Diagrama de transições de estados. (Fogel et al. [2007])*

Os atratores e as bacias de atração são informações sobre o modo funcional do sistema e determinam como o sistema se mantém no tempo, inclusive com relação a estabilidade, assim, uma questão importante que pode ser respondida utilizando estas informações é se a rede é estável ou não. Segundo Kauffman (Kauffman [1993], p191) e Li *et al.* [2004], redes estáveis são aquelas que possuem um pequeno número de atratores e um deles com a bacia de atração grande. Porém, apesar da bacia ser grande, apenas um pequeno número de estados faz parte do atrator. Os estados que não fazem parte do atrator, não são biologicamente significativos. Por outro lado, os estados que fazem parte do atrator podem representar um tipo de célula ou um estágio celular (diferenciação, proliferação ou apoptose [1]), caracterizados por seu padrão recorrente de expressão gênica. No contexto biológico, as redes estáveis são de grande interesse, pois representam um comportamento esperado para um sistema biológico, onde o sistema, mesmo após receber influências internas ou perturbações do ambiente externo, após certo período de estabilização, mantém seu funcionamento normal. Isso quer dizer que caso o sistema sofra alguma perturbação, a existência de uma bacia de atração grande aumentará a probabilidade de o sistema ser levado para algum estado da mesma bacia de atração que se encontrava, e consequentemente, ao mesmo atrator. Assim, a estabilidade das redes gênicas, por sua vez, faz com que as células mantenham seu estado funcional mesmo quando submetidas a perturbações externas.

A análise da dinâmica de uma rede booleana consiste na simulação da rede para obtenção dos *atratores*. Outra informação relevante que pode ser obtida é o *tamanho* das bacias de atração. A simples enumeração de todos os atratores já é um problema complexo. A tarefa é computacionalmente difícil, visto o problema ser NP-completo, sendo que a adição de um único vértice na a rede dobra o número de estados da mesma. Por este desafio, vários trabalhos abordam a análise da dinâmica de redes booleanas, com a finalidade de encontrar os atratores e adicionalmente determinar o tamanho das bacias de atração. Podemos citar: Skodawessely e Klemm [2011]), Akutsu *et al.* [2008], Dubrova *et al.* [2005], Dubrova e Teslenko [2011], Müssel *et al.* [2010], de Jong *et al.* [2003] e Irons [2006]. Este é o problema alvo deste trabalho, sendo enunciado basicamente como segue:

- Dada uma rede booleana $G(V, F)$ encontrar os atratores e determinar o tamanho das bacias de atração.

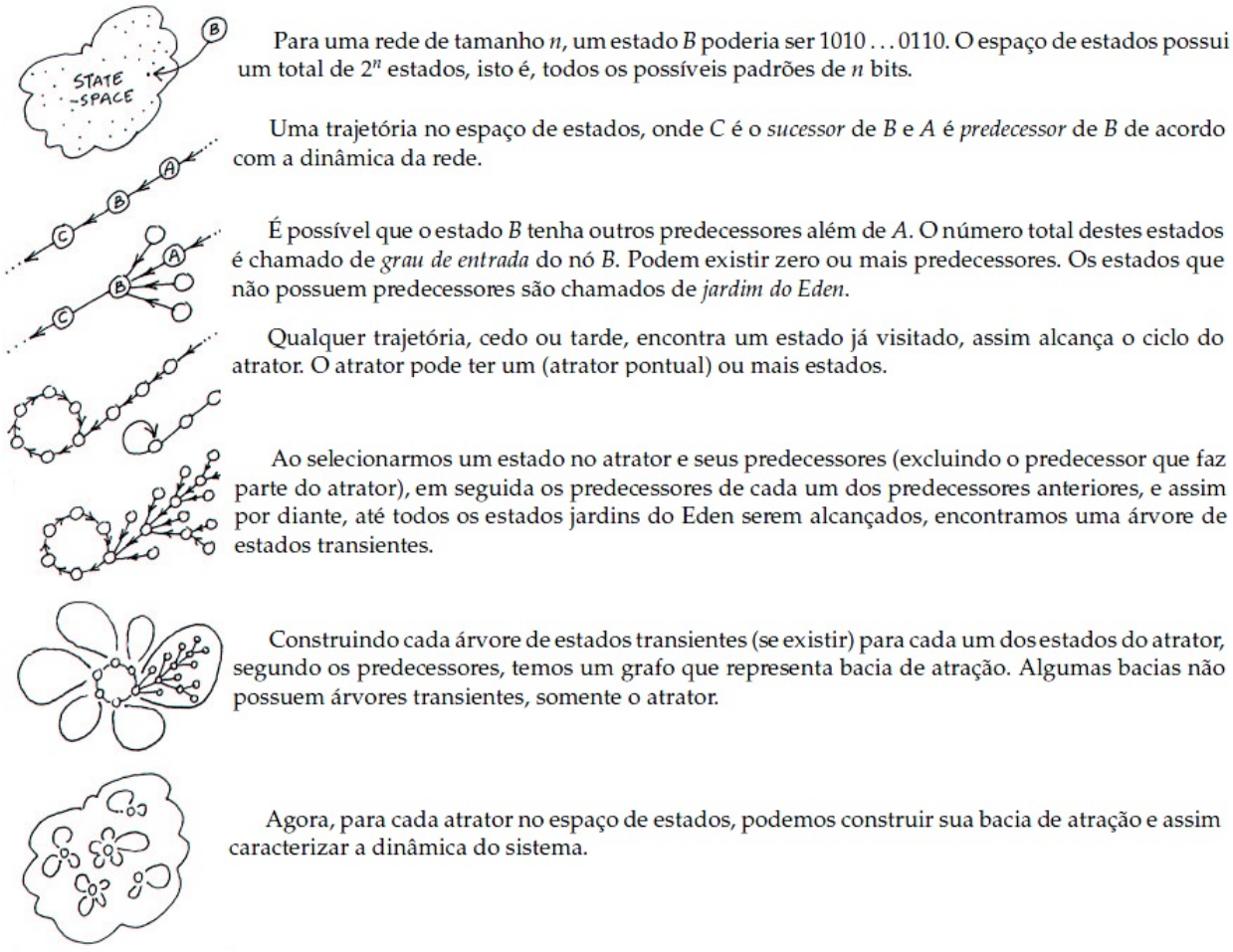
Alguns dos trabalhos citados realizam a análise da dinâmica através do problema de coloração em grafos, onde se deseja obter as componentes conexas de um grafo. No caso do grafo de sequência de estados, cada componente conexa representa uma bacia de atração e dado qualquer estado desta componente, para identificação do atrator é necessário simplesmente a realização de uma única trajetória até o mesmo.

Wuensche (Wuensche [1998]) propõe a realização de caminhamento para frente para localização dos atratores e bacias de atração. O sistema é simulado obtendo-se trajetórias que levam até um atrator. Para cada um dos estados do atrator (chamamos de raiz), utilizando os predecessores (excluindo estados do próprio atrator), encontra-se uma árvore de estados, com iterações sucessivas através das conexões com predecessores de cada estado localizado, até encontrar estados que não possuam predecessores (chamados de jardim do Eden). Com isso, todos os estados da bacia de atração são localizados. As conexões encontradas representam o grafo de transições de estado e a dinâmica do sistema. A Figura 2.3 ilustra o processo. Uma abordagem semelhante, para uma solução mais eficiente, foi desenvolvida por Irons [2006] (somente identificação dos atratores - não realiza contagem do tamanho das bacias de atração), que faz uso de pequenas sub-redes em seu algoritmo.

Neste trabalho, a fim de utilizar abordagens de computação de alto desempenho, pretendemos utilizar um modelo específico de rede booleana visto no trabalho de Li *et al.* [2004] e utilizado para

[1] Tipo de morte celular.

modelar com êxito o ciclo da divisão celular da levedura. Este modelo restringe o espaço das funções, utilizando um limiar, a rede então é chamada de *Rede Booleana Limiarizada*, na qual não são consideradas todas as funções booleanas possíveis do espaço das funções. Este modelo será detalhado na próxima seção.



Para uma rede de tamanho n, um estado B poderia ser 1010 . . . 0110. O espaço de estados possui um total de 2^n estados, isto é, todos os possíveis padrões de n bits.

Uma trajetória no espaço de estados, onde C é o *sucessor* de B e A é *predecessor* de B de acordo com a dinâmica da rede.

É possível que o estado B tenha outros predecessores além de A. O número total destes estados é chamado de *grau de entrada* do nó B. Podem existir zero ou mais predecessores. Os estados que não possuem predecessores são chamados de *jardim do Eden*.

Qualquer trajetória, cedo ou tarde, encontra um estado já visitado, assim alcança o ciclo do atrator. O atrator pode ter um (atrator pontual) ou mais estados.

Ao selecionarmos um estado no atrator e seus predecessores (excluindo o predecessor que faz parte do atrator), em seguida os predecessores de cada um dos predecessores anteriores, e assim por diante, até todos os estados jardins do Eden serem alcançados, encontramos uma árvore de estados transientes.

Construindo cada árvore de estados transientes (se existir) para cada um dos estados do atrator, segundo os predecessores, temos um grafo que representa bacia de atração. Algumas bacias não possuem árvores transientes, somente o atrator.

Agora, para cada atrator no espaço de estados, podemos construir sua bacia de atração e assim caracterizar a dinâmica do sistema.

Figura 2.3: *Localização dos atratores e bacias de atração (adaptado de Wuensche [1998]).*

2.1.3 Redes Booleanas Limiarizadas

Na definição de rede booleana o conjunto de funções pode conter qualquer função $f_i, i = 1, ..., n$, tal que $f_i : \{0,1\}^{k_i}$ e $1 \leq k_i \leq n$. O número total de funções booleanas possíveis para um gene v_i é $2^{2^{k_i}}$.

Li *et al.* [2004] descreve os relacionamentos regulatórios através de uma matriz, chamada de *matriz de regulação*. Para a matriz $A_{n\text{x}n}$, cada entrada a_{ij} utiliza a seguinte convenção:

$$a_{ij} = \begin{cases} -1, & \text{para uma regulação negativa (inibição) de } v_i \text{ para } v_j \\ 1, & \text{para uma regulação positiva (ativação) de } v_i \text{ para } v_j \\ 0, & \text{não há influência de } v_i \text{ para } v_j \end{cases} \tag{2.3}$$

O valor de cada gene no próximo instante de tempo (próximo estado da rede) $v_i(t+1), i = 1, ..., n$, é definido de acordo com a Equação 2.4, utilizando os elementos da matriz A e os valores dos genes preditores $v_j(t)$, $j = 1, ..., n$. Assim,

$$v_i(t+1) = \begin{cases} 1 & \text{se} \quad \sum_j^n a_{ij} v_j(t) > 0 \\ 0 & \text{se} \quad \sum_j^n a_{ij} v_j(t) < 0 \\ v_i(t) & \text{se} \quad \sum_j^n a_{ij} v_j(t) = 0 \end{cases} \tag{2.4}$$

e chamamos $I = \sum_j^n a_{ij} v_j(t)$ de *entrada* (do inglês *input*) do gene v_i no tempo $t+1$. Desta forma uma rede booleana limiarizada é uma rede $G(V,F)$, onde o conjunto de funções booleanas F é definido de acordo com o valor de entrada dos genes, dado pela matriz A. Com isso, em termos gerais, podemos definir a *rede booleana limiarizada* em termos do conjunto vértices e da matriz de regulação, como $G(V,A)$, onde V é o conjunto de vértices $V = \{v_1, v_2, ..., v_n\}$, e $A_{n\text{x}n}$ é a matriz de regulação gênica para os n genes.

A Figura 2.4 mostra um exemplo da rede booleana limiarizada para $n = 4$ genes $\{v_1, \ldots, v_4\}$, onde temos a matriz de regulação gênica T e uma representação gráfica da rede. Podemos ver que uma aresta, de um gene v_i para v_j, com extremidade no formato de flecha, mostra uma interação positiva (*ativação*), que é representada pelos elementos $T_{ij} = 1$ em $T_{1,3}$ e $T_{3,4}$ da matriz de regulação. Já uma aresta com extremidade de barra, indica uma interação negativa (*inibição*), que é representada pelos elementos $T_{ij} = -1$ em $T_{4,1}$, $T_{2,2}$ e $T_{2,3}$ da matriz de regulação. Uma aresta com formato tracejado com extremidade de barra indica que o gene possui uma característica de *auto-degradação*, tendo mesmo comportamento de uma inibição, neste caso também teremos $T_{ij} = -1$. Para os casos em que não existe interação temos $T_{ij} = 0$.

Com relação à limiarização, é fácil notar que nem todas as funções booleanas podem ser representadas por este modelo. Um caso trivial é aquele em que todos os genes estão com valor 0 no instante de tempo t, ou seja $v_i = 0$ para $i = 1, \ldots, n$. Neste estado, não existe uma função $f(v_1, v_2, \ldots, v_n) = 1$ qualquer que seja a matriz A, assim nunca teremos $f(0, 0, \ldots, 0) = 1$.

Na rede booleana limiarizada, a transição de um estado $s(t) = \{v_1(t), v_2(t), \ldots, v_n(t)\}$, no tempo t, para $s(t+1) = \{v_1(t+1), v_2(t+1), \ldots, v_n(t+1)\}$, é dada pela obtenção dos valores de entrada dos genes de $s(t+1)$ seguido da limiarização utilizando a Equação 2.4. Por questões simbólicas,

$$\mathrm{T} = \begin{array}{c} \\ v_1 \\ v_2 \\ v_3 \\ v_4 \end{array} \begin{array}{c} \begin{array}{cccc} v_1 & v_2 & v_3 & v_4 \end{array} \\ \begin{pmatrix} 0 & 0 & 0 & -1 \\ 0 & -1 & 0 & 0 \\ 1 & -1 & 0 & 0 \\ 0 & 0 & 1 & 0 \end{pmatrix} \end{array}$$

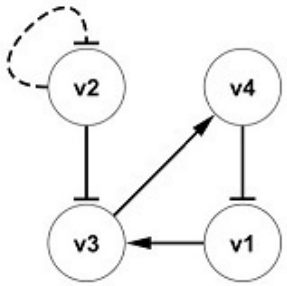


Figura 2.4: *Representação gráfica da **rede de regulação gênica**. À esquerda é mostrada a matriz de regulação gênica e à direita sua representação gráfica. Na representação gráfica, uma aresta com extremidade no formato de flexa indica uma regulação positiva, já uma aresta com formato de barra indica uma regulação negativa. A representação gráfica reflete a matriz de regulação. Por exemplo, $v_4 \dashv v_1$ indica uma regulação negativa, que é vista na matriz T_{14} como o valor -1.*

neste trabalho, representaremos essa operação como $A \bullet s(t)$, a qual é detalhada na Equação 2.5, onde é mostrada a multiplicação de uma matriz geral A por um vetor s de estado dos genes. Essa operação considera a limiarização (realizada no último passo após o símbolo $\cong$). Desta forma, uma transição de um estado $s(t)$ no tempo t para $s(t+1)$ no próximo instante de tempo, pode ser escrita simplesmente como $s(t+1) = A \bullet s(t)$. Assim temos:

$$\begin{aligned} s(t+1) &= A \bullet s(t) \qquad (2.5) \\ &\overset{A}{=} \begin{pmatrix} a_{11} & a_{12} & a_{13} & \dots & a_{1n} \\ a_{21} & a_{22} & a_{23} & \dots & a_{2n} \\ a_{31} & a_{32} & a_{33} & \dots & a_{3n} \\ \dots & \dots & \dots & \dots & \dots \\ a_{n1} & a_{n2} & a_{n3} & \dots & a_{nn} \end{pmatrix} \bullet \overset{\mathbf{s(t)}}{\begin{pmatrix} v_1(t) \\ v_2(t) \\ v_3(t) \\ \dots \\ v_n(t) \end{pmatrix}} \\ &\overset{\mathbf{I}}{=} \begin{pmatrix} a_{11}v_1(t) + a_{12}v_2(t) + a_{13}v_3(t) + \ldots + a_{1n}v_n(t) \\ a_{21}v_1(t) + a_{22}v_2(t) + a_{23}v_3(t) + \ldots + a_{2n}v_n(t) \\ a_{31}v_1(t) + a_{32}v_2(t) + a_{33}v_3(t) + \ldots + a_{3n}v_n(t) \\ \dots \\ a_{n1}v_1(t) + a_{n2}v_2(t) + a_{n3}v_3(t) + \ldots + a_{nn}v_n(t) \end{pmatrix} \cong \overset{\mathbf{s(t+1)}}{\begin{pmatrix} v_1(t+1) \\ v_2(t+1) \\ v_3(t+1) \\ \dots \\ v_n(t+1) \end{pmatrix}}, \end{aligned}$$

onde $\cong$ (aplicação do limiar - referente a Equação 2.4) é calculado para cada v_j, $i, j = 1, \ldots, n$, como,

$$v_j(t+1) = \begin{cases} 1 & \text{se} \quad I_j > 0 \\ 0 & \text{se} \quad I_j < 0 \\ v_i(t) & \text{se} \quad I_j = 0 \end{cases}$$

Considerando a matriz de regulação T do exemplo anterior (mostrada na Figura 2.4), podemos calcular todas as transições de estado, formando a dinâmica da rede, a qual é mostarda na Figura 2.5. Nesta figura, os atratores são formados pelos estados {0000}, {0001} e {0011}. Todos os outros estados são estados transientes.

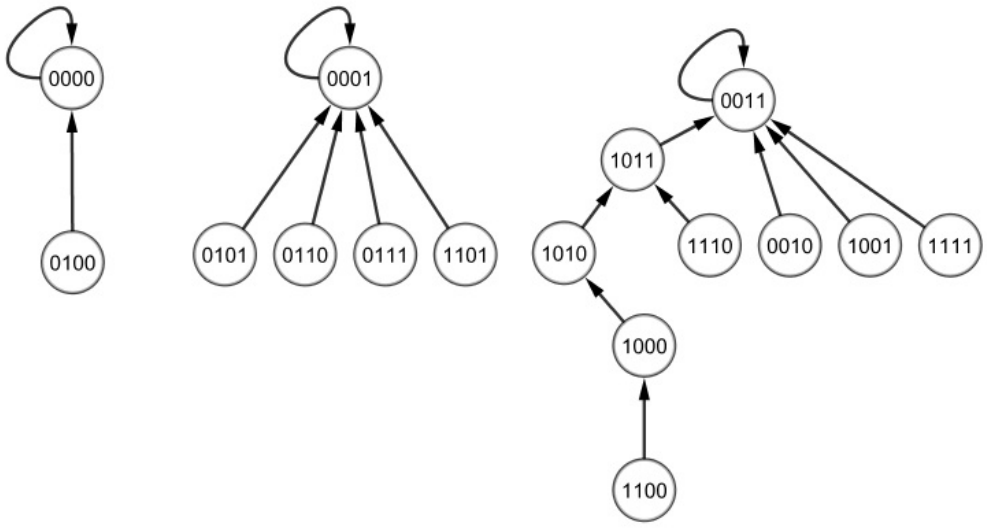


Figura 2.5: *Diagrama de sequência de estados da rede booleana limiarizada da Figura 2.4. Os atratores são os estados* {0000}, {0001} *e* {0011}.

Partindo do estado $s_1 = \{1100\}$ e fazendo $T \bullet s_1$ obtemos o próximo estado da rede, $s_2 = \{1000\}$, pois:

$$
\begin{aligned}
s(t+1) &= T \bullet s_1 \\
&= \begin{pmatrix} 0 & 0 & 0 & -1 \\ 0 & -1 & 0 & 0 \\ 1 & -1 & 0 & 0 \\ 0 & 0 & 1 & 0 \end{pmatrix} \bullet \begin{pmatrix} 1 \\ 1 \\ 0 \\ 0 \end{pmatrix} \\
&= \begin{pmatrix} 0 \\ -1 \\ 0 \\ 0 \end{pmatrix} \cong \begin{pmatrix} 1 \\ 0 \\ 0 \\ 0 \end{pmatrix}
\end{aligned}
$$

Vemos que o gene v_2 foi desativado (mudou do valor 1 para 0). Pela rede de regulação vemos que essa mudança é reflexo de uma auto-degradação ($v2 \dashv v2$).

2.1.4 Ferramentas Existentes

A identificação dos atratores de uma rede booleana é um desafio computacional o qual motiva a pesquisa de algoritmos para tal fim. Este problema é visto em diversos trabalhos, como Wuensche [1998], Irons [2006], Skodawessely e Klemm [2011], Dubrova e Teslenko [2011], Müssel *et al.* [2010], entre outros. Para um número elevado de genes as soluções são computacionalmente intensivas, face o crescimento exponencial do número de estados do sistema com o aumento do número de genes. Serão apresentados três métodos dos quais dois serão utilizados como base para comparação dos tempos de execução. O primeiro método consiste na utilização da solução geral (Hopfensitz *et al.* [2013]), onde enumera-se todas as trajetórias possíveis para contabilizar todos

atratores e tamanhos das bacias de atração. O segundo método é baseado no problema de Satisfatibilidade Booleana (Dubrova e Teslenko [2011]), o qual não computa o tamanho das bacias de atração, porém, se mostra como boa alternativa para busca de atratores em redes grandes com grau médio de entrada baixo.

Identificação de Atratores e Tamanho das Bacias de Atração - Solução Geral

Para obtenção dos atratores e tamanho das bacias de atração, a solução geral é para cada estado do espaço de estados, tomado como estado inicial, realiza-se uma trajetória até a identificação de um ciclo, ou seja o atrator. O estado inicial é marcado com a identificação do atrator e a frequência com que o atrator ocorre é armazenada. O processo termina quando todos os estados têm seu atrator identificado. A frequência com que cada atrator ocorre nos dá o tamanho da respectiva bacia de atração, do atrator em questão.

Identificação de Atratores baseado no problema de Satisfatibilidade Booleana - SAT

Utilizando o modelo genérico de redes booleanas o trabalho de Dubrova e Teslenko [2011] propõe uma solução promissora para obtenção dos atratores, porém **não** realiza a contagem do tamanho das bacias de atração. A solução apresentada bons resultados para redes com número elevado de genes e baixo grau de entrada para os genes, conseguindo manipular redes de ate 7000 genes, com grau de entrada máximo igual 2. A solução é baseada em um resolvedor SAT (do inglês *Satisfiability Problem* - ou Problema de Satisfação Booleana) e não é capaz de resolver em tempo hábil todas as instâncias de problemas. Porém uma análise estatística mostra que em média grande parte das instâncias de problemas (geradas de forma aleatória) é resolvida em tempo hábil (máximo de 10 minutos).

Para solucionar um SAT é necessário avaliar se existe uma solução que torne factível uma dada fórmula proposicional (expressão booleana), determinando assim se existe uma interpretação satisfatória para a mesma, ou seja se existe uma combinação de valores para suas variáveis a fim de satisfazer como verdadeira a expressão em questão. SAT foi o primeiro exemplo de problema NP-completo, mesmo com essa característica, alguns algoritmos, os chamados resolvedores SAT (SAT *solvers*), resolvem com eficiência um grande conjunto de instâncias de problemas. Não entraremos em detalhes sobre a implementação destes resolvedores.

Para modelar a dinâmica da rede utilizando SAT, uma trajetória de p estados é construída como uma fórmula proposicional, contendo um conjunto de cláusulas, formando assim o problema SAT básico ser resolvido. Nesta trajetória, cada uma destas cláusulas é composta pela a conjunção ("AND") das funções booleanas de cada gene.

Assim, para uma rede booleana de n genes, com as funções booleanas $f_1, f_2, \ldots, f_n$, uma trajetória de p estados $x(t)$, $t = 1 \ldots p$, é dada por $x(1), x(2) \ldots x(p)$. Esta trajetória é formulada como um conjunto de p cláusulas, cada uma contendo n variáveis, uma para cada gene, ou seja, para cada cláusula i o valor do gene j é assinalado na variável v_{ij}. Neste problema temos um total de $n * p$ variáveis sendo que sua solução é uma trajetória na rede booleana de tamanho p. Esta formulação é chamada de *relação de transição*, sendo mesma construída de forma incremental, com cada nova cláusula definida de acordo com a cláusula do instante de tempo anterior.

Podemos formular uma cláusula C_i para o instante de tempo t da seguinte forma:

$$\overbrace{v_{i1} \leftrightarrow f_1(t-1) \text{ and } v_{i2} \leftrightarrow f_2(t-1) \text{ and } \ldots \text{ and } v_{in} \leftrightarrow f_n(t-1)}^{C_i}$$

Então para p instantes de tempo, a relação de transição representa a seguinte fórmula proposicional:

$$C_1 \text{ and } C_2 \text{ and } \ldots \text{ and } C_p$$

No algoritmo, o resolvedor SAT é utilizado para verificar se esta fórmula possui solução factível. Em caso positivo é analisado se na solução existe um atrator (existência de ciclo). Em caso negativo a relação de transição tem seu tamanho dobrado. O atrator é identificado quando ocorre a repetição de um estado na solução encontrada. Cada atrator é adicionado como restrição na relação de transição atual, o que faz com que o resolvedor SAT busque por novas soluções, as quais o atrator em questão não faz parte, a fim de encontrar outros atratores. O processo é repetido até não haver mais solução para a relação de transição em questão.

A implementação utiliza o solver *MiniSat* [2] e está disponível em *http://web.it.kth.se/ dubrova/bns.html*.

[2] http://www.minisat.se/

2.2 Computação Paralela

O problema de determinar os atratores e efetuar a contagem do tamanho das bacias de atração apresenta características que possibilitam a utilização de técnicas de programação paralela. Especificamente para o modelo de rede booleana limiarizada, apresentado na Seção 2.1.3, face as características matriciais deste modelo, propõe-se a utilização de GPUs, como processadores auxiliares, a fim de utilizar seu alto poder computacional e de paralelismo, tornando possível a análise da dinâmica para redes com maior número de genes em menor tempo de processamento. Neste trabalho foram realizadas duas implementações GPUs como processadores auxiliares. A primeira é baseada em um algoritmo de componentes conexas em grafos. Já a segunda, realiza trajetórias em paralelo, inclusive utilizando múltiplas GPUs. O software desenvolvido, além de permitir ser utilizado como biblioteca, também permite a execução pela web. Para realizar a implementação em GPU é necessário estabelecer alguns conceitos de computação paralela e programação em GPU os quais serão abordados nesta seção.

Tradicionalmente, softwares eram escritos para serem executados de forma sequencial. Neste modelo de execução, a implementação de algoritmos é feita através de instruções executadas em série por uma Unidade Central de Processamento (*Central Processing Unit* - CPU) de um computador. Somente uma instrução é ser executada por vez, após sua execução, a próxima é então executada. Por outro lado, a computação paralela faz uso de múltiplos elementos de processamento. Isso é possível ao quebrar um problema em partes independentes de forma que cada elemento de processamento possa executar sua parte simultaneamente com outros. Os elementos de processamento podem ser diversos e incluir recursos como um único computador com múltiplos processadores, diversos computadores em rede, *hardware* especializado, etc.

Tal como em outras áreas da ciência da computação, a história da computação paralela não poderia ser diferente, ela surgiu em decorrência de desenvolvimentos marcantes verificados em *hardware*. Dentre os principais precursores desse tipo de programação, destacam-se os sistemas operacionais. Com a importante independência dos controles de dispositivos, alcançada na década de 1960, o processador central de um computador deixou de interagir diretamente com os dispositivos de entrada e saída, e passou a fazê-lo através de interfaces especializadas. Desta forma, pôde-se constatar que a utilização do processador central poderia ser consideravelmente otimizada, utilizando-se os seus tempos ociosos, decorrentes da espera pelas realizações das operações independentes de entrada e saída. Esses intervalos de tempo ociosos poderiam ser, por exemplo, utilizados no processamento de outros programas. Então os sistemas operacionais começaram a ser projetados como uma coleção de processos, em um ambiente em que as tarefas de usuários conviveriam harmoniosamente com os processos gerentes de dispositivos e os demais processos do sistema operacional (Silberschatz e Peterson [1994] e Tanenbaum [1992]). Em sistemas dessa natureza, onde um único processador é capaz de processar mais de um programa simultaneamente, conhecidos como sistemas multiprogramados, os processos são executados em fatias de tempo, que o processador dedica de forma intercalada.

A computação paralela tornou-se essencial no projeto de muitos outros sistemas computacionais, seja pela natureza inerentemente paralela apresentada pelo problema a ser resolvido, seja em ganho na eficiência do tempo de processamento, ou mesmo na melhor estruturação do sistema. Como nem todos os problemas são desta natureza, não é possível paralelizar todas as partes de um sistema, foca-se então nos pontos chave onde existe a necessidade de otimização do tempo da computação para um problema específico, primeiramente, verificando se o mesmo possui natureza inerentemente paralela. Uma análise de várias classes de problemas computacionais é feita por Asanovic (Asanovic *et al.* [2006]) quanto à possibilidade de paralelismo, levando em consideração diversos fatores que capturam requisitos destes problemas independentemente de sua implementação, por exemplo, problemas como os de álgebra linear densa (multiplicação de matrizes ou

vetores densos), métodos espectrais (como Transformada Fourier), métodos de Monte Carlo (onde os cálculos dependem de resultados estatísticos de repetidas tentativas aleatórias) e em especial, na visão deste trabalho, estão travessia em grafos (envolvendo tabelas de transição) e máquinas de estado finito (das quais algumas podem ser decompostas e podem ser processadas em paralelo).

Sub-tarefas de um programa paralelo são frequentemente chamadas threads. As threads de um programa executam suas instruções de forma independente e concorrente, podendo ser executadas em qualquer ordem no tempo. A implementação de threads e processos difere de um sistema operacional para outro. Uma thread é um "processo leve", em geral, contida em um processo, assim múltiplas threads podem existir no mesmo processo e compartilhar recursos tais como memória diretamente, enquanto diferentes processos possuem as áreas de memória reservadas e não realizam este compartilhamento de forma direta, sendo necessário o uso de protocolos específicos, por exemplo *Inter-Process Communication* (IPC) (Free Software Foundation, Inc [2008]), para realizar tal tarefa. Para as threads o compartilhamento é possível, de forma direta e transparente, pois todas as threads compartilham o mesmo contexto de execução automaticamente, assim sendo, possuem o mesmo conjunto de instruções e endereçamento de memória. A Figura 2.6 ilustra o conceito de threads e processos.

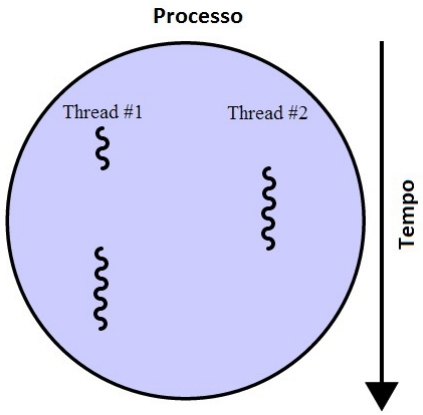


Figura 2.6: *Modelo de execução de múltiplas threads, as quais executam sobre um mesmo processo. A thread 1 pode executar o mesmo conjunto de instruções da thread 2 ou instruções diferentes. Ambas compartilham o mesmo contexto de execução do processo, sendo memória um recurso possível de ser compartilhado entre as duas linhas de execução (imagem adaptada de Wikipedia [2013]).*

2.2.1 Programação Geral para GPU

As GPUs foram concebidas para computação intensiva e altamente paralela, que são as necessidades da computação gráfica (NVIDIA [2007], p. 2). Para isto, a GPU realiza a conversão de uma descrição de cena em uma imagem. Este processo é dividido em várias etapas, como em uma linha de montagem, chamada de *pipeline gráfico*, constituído de circuitos integrados específicos. Nas primeiras GPUs estes *pipelines* possuíam uma função fixa, eram configuráveis, porém não acessíveis ao programador (Kirk e Hwu [2010]).

Uma das técnicas de programação paralela é através da utilização de GPU, chamada de GPGPU (*General Purpose Computing on Graphics Processing Units*). Esta técnica tem revitalizado o estudo de computação paralela provendo uma arquitetura poderosa a um custo relativamente baixo. Uma das implementações mais difundidas é CUDA (*Compute Unified Device Architecture*), desenvolvida pela NVIDIA. Outra opção é a *OpenCL* ™(*Open Computing Language*), a qual provê também uma API ™ de programação para GPGPU, suportando outros distribuidores de GPU, como AMD ©[3].

[3] www.amd.com

A OpenCL trás uma padronização para programação paralela em sistemas heterogêneos (CPU e GPU), possibilitando o software desenvolvido ser portável e eficiente através de diferentes plataformas (Khronos Group [2012]).

A partir do final dos anos 90, o *hardware* tornou-se cada vez mais programável. Em 2001, as GPUs passaram de um pipeline de função fixa para geração de imagens para um pipeline altamente programável, aumentando a capacidade de gerar efeitos visuais customizados. Foi uma questão de tempo até perceber-se a possibilidade de utilizar esse hardware para executar processamento numérico e científico. Iniciando em 2003 e utilizando APIs como *DirectX* (API gráfica para Windows ™criada pela Microsoft ©) e o *OpenGL* ™ (*Open Graphics Library* - API gráfica multiplataforma aberta), alguns programadores conseguiram utilizar os recursos computacionais das GPUs para fins diferentes de somente renderização de imagens. Porém, nessa época, escrever um programa para utilizar os recursos disponíveis na placa gráfica era complicado, pois o programador precisava conhecer profundamente o *hardware* e teria que definir o problema em termos de coordenadas de vértices, texturas e programas de renderização. Além disso, não havia recursos para as leituras e escritas aleatórias na memória (Sanders e Kandrot [2010], p. 5).

Vemos assim, que uma das maiores barreiras no inicio da utilização de GPUs, para programação geral, era a necessidade de descrição dos problemas através de API gráficas, com isso, alguns fabricantes desenvolveram plataformas específicas para realização da programação geral em GPU de forma genérica. Essa foi uma das motivações para surgimento de CUDA, a qual será detalhada em seguida.

2.2.2 CUDA - *Compute Unified Device Architecture*

Em novembro de 2006, a NVIDIA introduziu CUDA, uma arquitetura de computação paralela de propósito geral, com um novo modelo de programação paralela e conjunto de instruções, que aproveita o mecanismo de paralelismo da GPU para resolver problemas computacionalmente complexos de maneira mais eficiente do que em CPU (NVIDIA [2011], p. 3).

O sistema de execução CUDA consiste de um computador hospedeiro (*host*) com uma CPU tradicional que efetua o processamento central, e uma ou mais GPUs (*device*). As fases do programa que exibem pouco ou nenhum paralelismo sobre os dados são executados no *host*. Já as que exibem rico grau de paralelismo são executadas no *device* (Kirk e Hwu [2010], p. 42). A GPU executa milhares de threads, que são gerenciadas de maneira que sejam alocadas paralelamente e o usuário usufrua dessa característica para aproveitar o alto grau de paralelismo sobre os dados.

O programa que é executado na GPU é chamado de *kernel*. Um *kernel* possui o código fonte que especifica a parte do algoritmo que se deseja a paralelização. O *kernel* deve possuir uma característica de processamento paralelo e não deve possuir dependência da ordem de execução. No modelo de programação em CUDA cada instância do *kernel* (uma thread "leve") realiza sua computação em dados diferentes, a alocação da execução é feita em grupos de threads chamados de *blocos*. Um conjunto de blocos especifica um *grid*. É essencial notar que as threads CUDA são muito mais “leves” que threads de CPU e pode-se assumir que essas threads levam poucos ciclos de *clock* para serem geradas dado o eficiente suporte do *hardware* da GPU. Em contraste, as threads de CPU tipicamente requerem milhares de ciclos de *clock* para serem geradas. A Figura 2.7 ilustra o processo de execução típico em CUDA, onde temos duas chamadas de *kernel*, A e B gerando dois *grids*, onde cada um contém *nBIK* blocos de *nTid* threads.

A execução inicia no *host*, o qual efetua configuração de parâmetros do *kernel*, como número de threads, alocação de memória e transferência dos dados para GPU. O *kernel* então é inicializado e a execução move-se para a GPU. Esta execução é feita de forma assíncrona, isto é, a thread de CPU que executa o *kernel* não fica aguardando sua finalização, para isto, existem mecanismos especí-

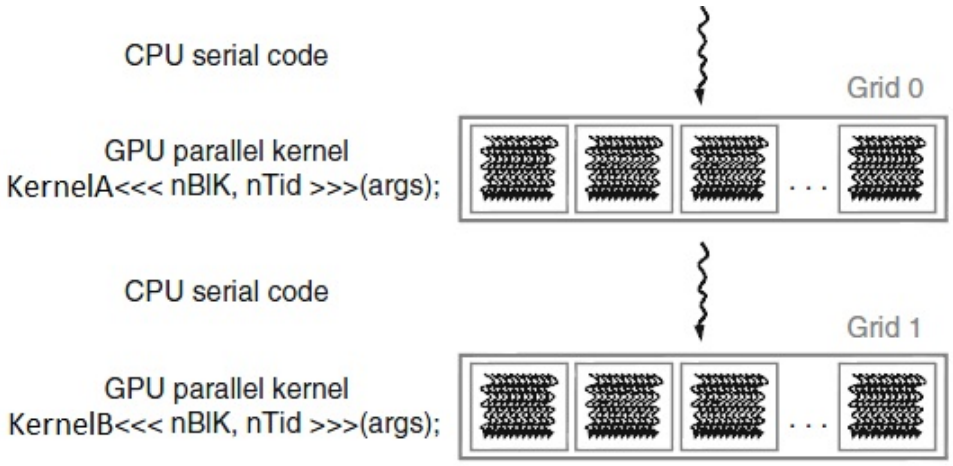


Figura 2.7: *Modelo de execução CUDA (adaptado de Kirk e Hwu [2010], p. 42).*

ficos de bloqueio e sincronização. Na GPU, um vasto número de threads são geradas, segundo a configuração estabelecida. Cada uma destas, executará o código especificado para o *kernel* (em uma extensão da linguagem C, provida pela API de programação CUDA) sobre dados específicos.

Em resumo, o fluxo de execução é dado pelos seguintes passos:

- Configuração dos parâmetros de *kernel* e transferência dos dados a serem utilizados da memória do computador principal para a memória da GPU;
- Execução do programa na GPU (os resultados ficam na memória da própria GPU);
- Transferência dos resultados da memória da GPU para a memória do computador principal;
- O processamento do programa principal continua normalmente, é feita liberação de recursos alocados.

A arquitetura CUDA é baseada em *Streaming Multiprocessors* (SM). Um SM é capaz de executar milhares de threads concorrentemente. Para gerenciar este alto número de threads é utilizada uma arquitetura SIMT (Single Instruction Multiple Threads - Instrução Única Múltiplas Threads), onde os dados são processados de forma similar ao modelo SIMD (Single Instruction Multiple Data - Instrução Única Múltiplos Dados). O modelo de execução SIMT, utiliza múltiplas threads e cada uma executa o mesmo programa sobre dados diferentes ao mesmo tempo. A similaridade está na organização do vetor de dados, em que uma única instrução controla múltiplos elementos de processamento. A diferença está na exposição do vetor de dados, onde SIMD está relacionada com software, enquanto que SIMT especifica o comportamento da execução de uma única thread, oferecendo liberdade aos programadores para escrever código paralelo no nível de threads (NVIDIA [2011], p. 62).

Um *Multiprocessor* é responsável por criar, gerenciar, alocar e executar threads em grupos de 32 threads paralelas, chamadas *warps*. O grupo de threads que compõe um *warp* iniciam juntas do mesmo endereçamento, mas possuem seu próprio contador de instruções e registrador de estado, sendo assim, são independentes para realizar fluxos diferentes de execução. Quando um bloco é designado a ser executado em um multiprocessor, este particiona o bloco em *warps*, que são alocados por um gerenciador de *warps* para execução. A maneira como um bloco é particionado em *warps* é sempre a mesma: cada *warp* contém threads de identificadores (para cada thread é dado um identificador único - `threadId`) consecutivos e crescentes, com o primeiro *warp* contendo a thread de identificador igual a 0 (NVIDIA [2011], p. 61).

Um *warp* executa uma instrução por vez. A eficiência máxima é alcançada quando todas as 32 threads de um *warp* entram em concordância sobre o fluxo de execução a ser realizado, porém, podem haver divergências condicionais devido à alguma peculiaridade dos dados. O *warp* então serializa a execução de cada fluxo divergente, desabilitando threads que não estão no fluxo em questão. Ao final as threads convergem de volta para o mesmo fluxo de execução. A divergência de fluxo é interna ao *warp*, sendo assim diferentes *warps* não dependem dos fluxos comuns ou divergentes para atingir máxima eficiência.

Compilação

O código fonte de um programa em CUDA possui uma porção de código C, para ser executado no computador hospedeiro, e outra porção para ser executado na GPU. Por esta razão o processo de compilação deve ser diferenciado e consiste de fases separadas de compilação.

O código a ser executado na GPU é compilado por um compilador proprietário da NVIDIA (chamado NVCC - *NVIDIA Cuda Compiler*), que compila também o código a ser executado no computador hospedeiro utilizando um compilador geral C/C++, e em seguida efetua a ligação entre os dois. Na fase de ligação, biblioteca dinâmica CUDA é adicionada para suporte às chamadas de funções explícitas da GPU, como alocação de memória e transferência de dados (NVIDIA [2010]).

O NVCC facilita na separação, compilação e união dos objetos compilados para o código fonte do *host* e da GPU, funcionando similarmente ao compilador GNU GCC (*GNU Compiler Collection* - Coleção de Compiladores GNU), o qual é utilizado para compilação de código nas plataformas Linux. Em plataformas Windows, o compilador utilizado é o *Microsoft Visual Studio*. A Figura 2.8 ilustra o processo.

O compilador NVCC recebe o código fonte de uma aplicação em CUDA (combinação de código para alvos de execução CPU e GPU) como entrada. Neste processo, os códigos são separados e compilados através de compiladores específicos, gerando arquivos objeto para cada um dos alvos de execução. Ao final, é realizada a ligação de ambos arquivos objetos, inclusive com bibliotecas dinâmicas necessárias, gerando um executável.

Modelo de memória e transferência de dados

A memória da GPU é separada da memória do *host* e CUDA oferece uma API de funções para realizar transferência de dados da memória do *host* para memória da GPU (implementada tipicamente como DRAM - *Dynamic Random Access Memory*) e vice versa. Para executar um *kernel* é necessário alocar a memória da GPU e transferir os dados pertinentes. Da mesma forma, após a execução, os dados precisam ser transferidos de volta para a memória do *host* e os recursos alocados liberados.

Os espaços de memória da GPU são:

1. **Memória Global:** Memória principal e de maior capacidade de armazenamento.
2. **Memória Local:** Memória disponível para cada thread.
3. **Memória de Constantes e de Textura:** Memória específica utilizada para armazenar constantes e formatos específicos de dados.
4. **Memória Compartilhada:** Memória compartilhada entre as threads de um bloco.
5. **Registradores:** Memória de acesso de alta velocidade interna ao chip da GPU.

As threads podem acessar dados nestes diferentes espaços de memória durante sua execução. A eficiência e velocidade do acesso aos diferentes espaços é vista de forma decrescente dos itens 1 à 5, isto é, o acesso à memória global possui maior latência, enquanto o acesso aos registradores,

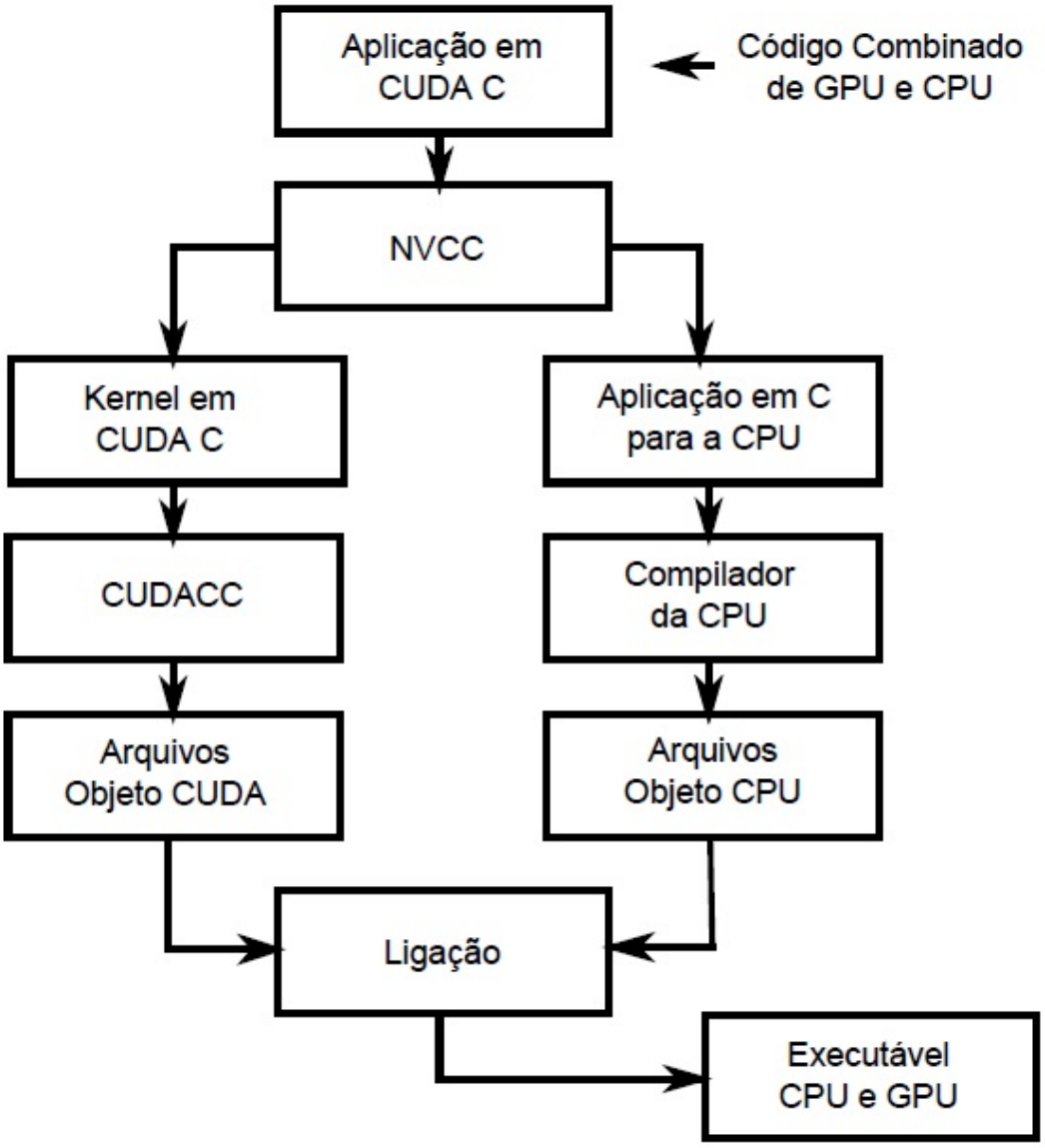
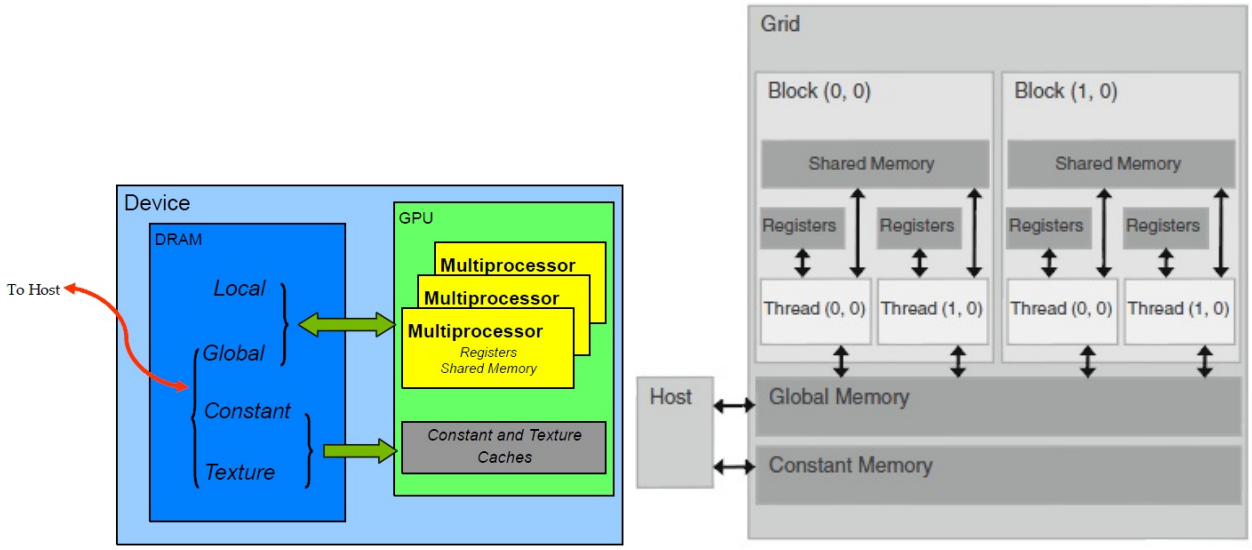


Figura 2.8: *Fases da compilação em CUDA (Santos [2012], p. 28).*

possui menor latência. Os espaços de memória são ilustrados nas figuras 2.9a e 2.9b.

Cada thread possui sua memória local (privada). Cada bloco de threads, possui uma porção de memória compartilhada, visível para todas as threads do bloco e com o mesmo ciclo de vida que o bloco. Todas a threads possuem acesso a mesma memória global. Existem ainda dois espaços de memória de somente leitura, que utilizam *cache*, acessíveis por todas as threads, a memória de constantes e de textura, as quais são otimizadas para diferentes utilidades. Além disso, a memória de textura também oferece diferentes modos de endereçamento e filtros para alguns formatos de dados específicos, por exemplo, de imagens. Os espaços de memória global, de constantes e textura possuem ciclo de vida por toda a execução do *kernel* (NVIDIA [2011], p. 10).

Registradores e memória compartilhada são memórias que residem internas à GPU. Variáveis que residem nestes tipos de memória podem ser acessadas com altíssima velocidade e alto grau de paralelismo. Registradores são alocados individualmente por thread e cada uma pode acessar somente seus próprios registradores (Kirk e Hwu [2010], p. 79).



(a) *Nível de Hardware. Cada Multiprocessor possui registradores e memória compartilhada internas ao chip da GPU. É realizada cópia do host para DRAM e vice-versa (NVIDIA [2012b], p. 31)*

(b) *Nível de Software. Todas as threads de um grid possuem acesso à memoria global. Memória compartilhada é acessível pelas threads de mesmo bloco. Registradores são acessíveis por cada thread. (Kirk e Hwu [2010], p. 79)*

Figura 2.9: *Ilustração do modelo de memória CUDA.*

A Tabela 2.1 apresenta a sintaxe CUDA para declaração de variáveis para os vários tipos de memória, incluindo o escopo e tempo de vida da mesma. O escopo identifica o conjunto de threads que podem acessar a variável, isto é, uma única thread, as threads de um bloco ou as threads de um *grid*. O tempo de vida especifica a duração de tempo que uma variável está disponível para uso, podendo ser somente interno à invocação do *kernel* ou por toda a aplicação.

Declaração	Memória	Escopo	Tempo de Vida
Variáveis automáticas (com exceção de arrays)	Registrador e Local	Thread	Kernel
Arrays automáticos	Local	Thread	Kernel
__device__, __shared__, int SharedVar	Compartilhada	Block	Kernel
__device__, int GlobalVar	Global	Grid	Aplicação
__device__, __constant__, int ConstVar	Constante	Grid	Aplicação

Tabela 2.1: *Qualificadores de variáveis em CUDA.*

Todas as variáveis automáticas (com exceção de arrays) declaradas nas funções de *kernel* são alocadas em registradores ou na memória local (no caso de exceder a capacidade dos registradores). Seus escopos são individuais por thread, sendo que cada uma recebe uma cópia da variável em questão. No caso de variáveis alocadas em registradores o acesso realizado é extremamente rápido, podendo ser realizado com alto grau de paralelismo, porém deve ser realizado com cautela para não exceder o limite de capacidade de armazenamento em registradores. Neste caso, é utilizada a memória local, cuja latência de acesso é a mesma da memória global. Variáveis automáticas para armazenar *arrays* não são armazenadas em registradores, ao invés disso, elas são armazenadas na memória local ou global e assim ficam sujeitas a um acesso não tão rápido. O escopo destes *arrays* também é individual por thread.

Em seguida veremos particularidades da Memória Global, da Memória Local e da Memória Compartilhada.

Memória Global

A memória global normalmente é utilizada armazenar a massa de dados necessária para execução de um *kernel*. Estes dados são provindos da memória do computador hospedeiro. A tecnologia atual utilizada para realizar esta transferência é a *PCI Express*^{TM} (Peripheral Component Interconnect Express, ou seja, Interconexão Expressa de Componente Periférico). A arquitetura da placa gráfica G80, que introduziu a arquitetura CUDA tinha 86.4 GB/s de largura de banda de memória, mais 8 GB/s de largura de banda de comunicação com a CPU, na qual os dados podem ser transferidos a uma taxa de 4 GB/s em duas vias (do computador hospedeiro para a GPU e da GPU para o computador hospedeiro). Vemos que originalmente a largura de banda de comunicação era muito menor que a largura de banda de memória e talvez seja uma limitação. Entretanto a largura de banda do *PCI Express* é comparável à do barramento central entre CPU e o sistema de memória de um computador convencional, logo, não é realmente uma limitação como vista inicialmente. No futuro, espera-se que a largura de banda de comunicação cresça da mesma forma que a largura de banda do barramento central com a CPU (NVIDIA [2011], p. 8).

CUDA fornece mecanismos de transferência de dados através de funções específicas. Desta forma a realização desta tarefa torna-se transparente ao programador. As funções de transferência de dados permitem realizar a alocação de memória na GPU e também a transferência, tanto do computador hospedeiro para GPU como da GPU para o computador hospedeiro. Duas funções básicas para realizar esta transferência são:

- `cudaMalloc(mem_device, mem_size)`
- `cudaMemcpy(mem_host, mem_device, mem_size, transfer_type)`

A função `cudaMalloc` efetua a alocação de memória na GPU (NVIDIA [2012a]). Têm como parâmetros uma variável que irá armazenar uma referência para a memória alocada (`mem_device`) e a quantidade de memória a ser alocada (`mem_size`). Esta referência é utilizada posteriormente na realização da transferência de memória e execução do *kernel*.

A função `cudaMemcpy` efetua a transferência de memória (NVIDIA [2012a]). Têm como parâmetros uma referência para a memória do computador hospedeiro (`mem_host`), uma referência para a memória de GPU (`mem_device`), alocada utilizando `cudaMalloc`, a quantidade de memória a ser transferida (`mem_size`) e o tipo de transferência (`transfer_type`), que indica a direção da transferência, isto é, se é do computador hospedeiro para a GPU ou o contrário.

Dizemos que são funções básicas pois existem outras abordagens visando otimizações na realização destas transferências. Uma delas é utilização de *Page Locked* na memória do computador hospedeiro. *Page Locked* significa "página travada", e faz com que determinada página de memória seja impedida de ser utilizada como memória virtual. O uso dessa técnica permite que a própria memória do hospedeiro seja mapeada como memória da GPU, eliminando a transferência de dados. O inconveniente é a pequena disponibilidade de memória a ser travada e eventual diminuição de desempenho no hospedeiro. Essa diminuição de desempenho é devida à retirada da memória travada do total de memória disponível para paginação do sistema de memória virtual do hospedeiro (NVIDIA [2011], p. 28-29).

Outra forma de melhorar o desempenho das transferências entre GPU e hospedeiro é a execução concorrente assíncrona. Ela permite que o controle retorne para o hospedeiro antes do final da execução do *kernel*. Algumas GPUs possuem ainda o recurso de transferência entre a memória da página travada e a GPU de forma concorrentemente com a execução do *kernel* (NVIDIA, 2011b, p. 30), neste caso, maximizando ainda mais a performance. No caso de haver várias GPUs no mesmo hospedeiro, é possível ainda a cópia de dados entre as memórias dessas GPUs (NVIDIA [2011], p.

30-33)).

Memória Local

A memória local têm este nome pelo seu escopo ser local à thread, não pela localização física na placa gráfica, que é fora do chip. É utilizada somente para armazenar variáveis automáticas, o que é feito somente quando compilador NVCC identifica que não há espaço suficiente para armazenar variáveis nos registradores, por exemplo, quando as variáveis são grandes estruturas de dados ou *arrays* que consomem mais espaço do que o disponível. Esta memória reside na DRAM da GPU, então o acesso tem a mesma latência e baixa largura de banda que o acesso a memória global, desta forma, o acesso é tão caro quanto o acesso a memória global.

Memória Compartilhada

A memória compartilhada é um meio eficiente das threads cooperarem através do compartilhamento dos seus dados de entrada e resultados intermediários de seu trabalho. Por ser interno ao chip, possui alta largura de banda de acesso e menor latência que a memória global. Para alcançar esta alta largura de banda de para acesso concorrente, é dividida em módulos de tamanhos iguais, chamados bancos de memória, que podem ser acessados simultaneamente, assim qualquer carga ou armazenamento em n endereços que atinge n bancos de memória distintos podem ser realizados simultaneamente, produzindo uma largura de banda efetiva de acesso que é n vezes tão alta quanto a de um único acesso à um único banco (NVIDIA [2012b]).

A declaração de uma variável compartilhada é feita precedendo a declaração da variável com a palavra reservada `__shared__`, por exemplo:

```
__shared__ int count;
```

Tais declarações são normalmente definidas internamente à função de *kernel*. O escopo de uma variável compartilhada é de bloco, isto é, possui visibilidade para todas as threads de mesmo bloco. O ciclo de vida é dado pelo tempo de duração da execução do *kernel*, sendo que uma versão privada da variável compartilhada é criada para uso de cada bloco. Quando o *kernel* finaliza sua execução, o conteúdo da variável compartilhada deixa de existir. A memória compartilhada é utilizada para armazenar a porção de dados da memória global que é acessada de forma mais intensa durante alguma fase da execução do *kernel*, sendo uma das formas úteis para otimização de código, maximizando a largura de banda de acesso.

Especificação de *Kernels*

Cada thread executa em paralelo a função kernel, sendo que cada uma computa um conjunto de dados diferente. CUDA estende a linguagem C adicionando qualificadores, variáveis e sintaxe específica que permite a especificação de uma função como *kernel*. Para isto, utiliza-se o qualificador `__global__` , sendo que a sintaxe da chamada da função permite informar o número de threads executadas através do número de blocos e número de threads por bloco, como mostrado na Listagem 2.1, com relação a `nBlocks` e `nThreads` configurados entre << e >>, podendo ser do tipo de dados `int` (inteiro) ou `Dim3` (tipo de dado CUDA que permite a representação de até 3 dimensões), o que provê uma maneira natural de computar elementos em domínios como vetores, matrizes ou volumes.

A execução da linha 13 é realizada na GPU. Em tempo de compilação, internamente ao *kernel*, são disponibilizadas algumas variáveis. Desta forma, durante a execução os valores configurados e outras informações tornam-se acessíveis pelas threads, assim, as dimensões do *grid*, dimensões dos blocos, índice do bloco e índice da thread em questão podem ser utilizados. Para dimensões

do *grid* e de blocos temos as construções de código `gridDim` e `blockDim`. Já para os índices de bloco e thread temos `blockIdx` e `threadIdx`. Para acessar o índice atual do bloco pelas suas dimensões, são utilizadas as construções `blockIdx.x` e `blockIdx.y` respectivamente. Da mesma forma, para acessar os índices da thread pelas suas dimensões, utilizamos `threadIdx.x`, `threadIdx.y` e `threadIdx.z`.

```
1   __global__ kernelA(int paramIn[N][N], int paramOut[N][N]) {
2       int i = blockIdx.x * blockDim.x + threadIdx.x;
3       int j = blockIdx.y * blockDim.y + threadIdx.y;
4       ...
5       paramOut[i][j] = paramIn[i][j] * alpha;
6       ...
7   }
8
9   int main(){
10      ...
11      Dim3 nBlocks(3, 2);  // Numero de Blocos no Grid
12      Dim3 nThreads(4, 3); // Numero de Threads por bloco
13      kernelA<<nBlocks, nThreads>>(paramIn, paramOut);
14      ...
15  }
```

Listagem 2.1: *Exemplo de código CUDA C.*

A combinação de `blockIdx` e `threadIdx` identifica unicamente cada thread e é utilizada para realizar a organização do acesso aos dados ou executar dependências condicionais (Harish e Narayanan [2007]). O índice de uma thread está relacionado com seu identificador e podem ser obtidos de forma simples: Para um bloco de uma dimensão, o identificador e índice são iguais, portanto, igual a `threadIdx.x`. Para um bloco de duas dimensões (D_x, D_y), o identificador da thread de índice (x, y) é $(x + y * D_x)$. Para um bloco de três dimensões (D_x, D_y, D_z), o identificador da thread de índice (x, y, z) é $(x + y * D_x + z * D_x * D_y)$.

Nas linhas 2-3 é feito um mapeamento dos dados de entrada, na posição (i, j), para serem processados em determinada thread. O resultado é armazenado nos parâmetros de saída, que podem ser acessado novamente pela CPU, após a execução do *kernel*, através de transferência de memória. Uma ilustração da disposição dos blocos e threads pode ser vista na Figura 2.10, onde temos a mesma configuração da Listagem 2.1 e podemos notar que a thread de índice $(2, 1)$ no bloco $(1, 1)$ irá processar a posição $(6, 4)$ dos parâmetros de entrada.

Existem outros dois qualificadores: `__host__` e `__device__`. O primeiro define que código será executado e chamado no computador hospedeiro. Esse qualificador pode ser omitido. Já o segundo, informa que a função definida é uma função a ser executada na GPU. A diferença entre `__device__` e `__global__` está na hierarquia de chamada. Uma função que utiliza o qualificador `__global__` pode ser chamada do computador hospedeiro, enquanto que uma função que utiliza `__device__` só pode ser chamada por uma função `__global__` ou por outra função `__device__`.

Sincronização

Threads de um bloco podem cooperar compartilhando dados utilizando a memória compartilhada. Para tal, é necessário sincronizar sua execução para gerenciamento do acesso à memória. Mais precisamente, podem ser especificados pontos de sincronização no *kernel* efetuando uma chamada para a função `__syncthreads`. Esta função atua como uma barreira, a qual as bloqueia as threads para que sigam a execução a partir do mesmo ponto, portanto, deve ser executado por todas as threads do bloco (NVIDIA [2011], p. 10).

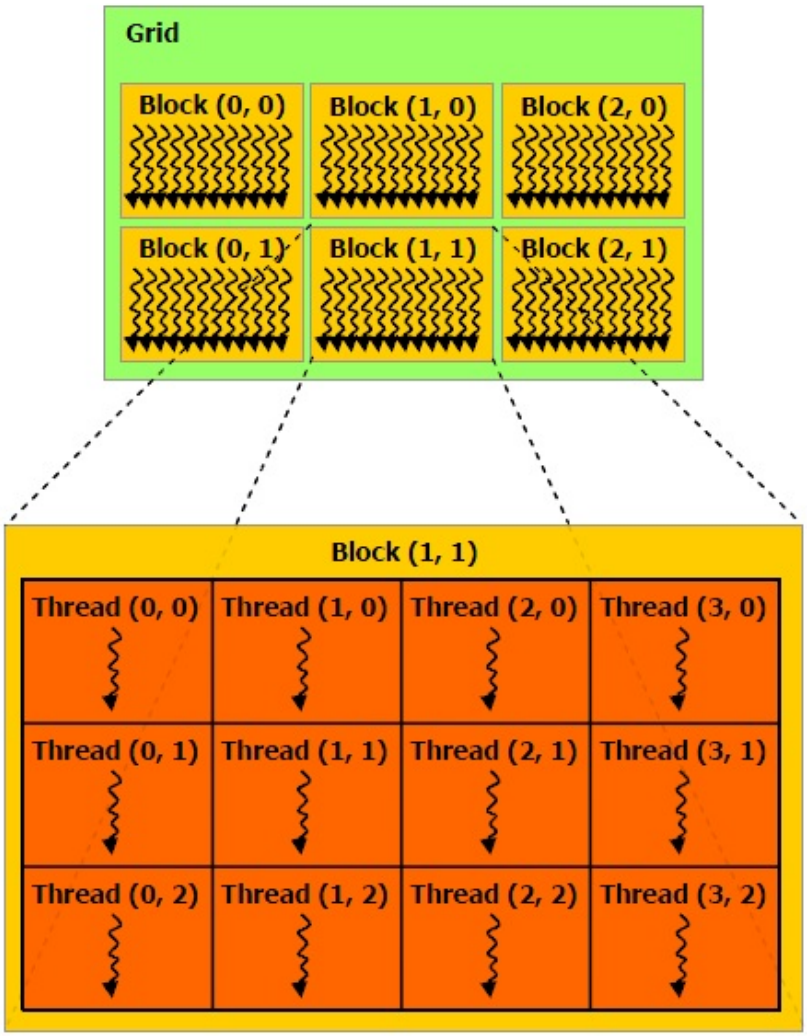


Figura 2.10: *Modelo de execução CUDA e organização dos índices dos blocos e threads (NVIDIA [2011], p. 9).*

Com relação ao fluxo de execução, caso seja realizado um fluxo condicional que contenha uma barreira de sincronização, todas as threads precisam executar o mesmo caminho de execução, caso não executem, pode haver espera indeterminada, fazendo com que a execução nunca finalize. Por exemplo, para um fluxo de execução `if-else`, caso exista um ponto de sincronização para cada caminho de execução, considera-se que as chamadas feitas para `__syncthreads` são independentes, e desta forma são dois pontos de sincronização diferentes. Se uma thread em um bloco executa o fluxo `if` e outra o `else`, aguardarão em diferentes barreiras de sincronização, e nunca continuarão a execução (Kirk e Hwu [2010], 68).

Considerações sobre desempenho

Um dos fatores importantes para se obter eficiência no desempenho da execução de um *kernel* está na configuração dos parâmetros de execução do mesmo, isto é, do número de blocos e threads utilizados. A correta configuração permite maximizar a ocupação dos processadores da GPU e assim utilizar todos os recursos de *hardware* disponíveis. Estabelecer estes parâmetros de forma correta é uma fase importante na especificação de um *kernel*. Para isto, é importante o entender o funcionamento da alocação das threads com relação aos *warps* e SMs.

O número máximo de threads e o número máximo de blocos que podem ser executadas por um SM variam dependendo do modelo da GPU, na qual possui também um número específico de

SMs disponíveis. O controle da distribuição de threads por SM é realizado pela GPU a qual utiliza como base os parâmetros de configuração para número de blocos e número de threads por bloco, do lançamento do *kernel*, para realizar esta distribuição, levando em consideração as limitações de recursos de cada SM. Uma aplicação que permite a configuração dos parâmetros de lançamento do *kernel* de forma genérica, ganha em flexibilidade e portabilidade entre diferentes GPUs, pois permite ajustes finos da execução para um *hardware* específico, sem necessidade de alteração do código fonte. Quando um *kernel* é executado, o sistema de execução CUDA gera o *grid* de threads correspondente e designa qual SM é responsável por cada bloco.

Por exemplo, a implementação da NVIDIA GT200, possui 30 *Streaming Multiprocessors*. No máximo 8 blocos podem ser designados para cada SM, sendo que cada SM pode executar no máximo 1024 threads. Estes limites definem os recursos de processamento disponíveis. A distribuição dos blocos nos SMs respeita o limite destes recursos, sendo que em situações cujo número de recursos é insuficiente, é realizada a redução do número de blocos automaticamente. Desta forma, no máximo 240 blocos (30 SM $*$8 blocos) podem ser simultaneamente executados, sendo que configuração de execução do *grid* pode conter um número muito maior que 240 blocos, que são distribuídos entre os recursos de processamento também de forma automática, divididos em unidades de 32 threads, formando um *warp*. O tamanho de um *warp* também varia entre implementações específicas de GPUs, e é a unidade básica para alocação de *threads* nos SMs. Assim, em termos de número máximo de threads, temos:

- 4 blocos de 256 *threads* (possível);
- 8 blocos de 256 *threads* (possível);
- 16 blocos de 64 *threads* (impossível, pois cada SM pode acomodar apenas 8 blocos).

Vemos que no máximo 30720 threads podem simultaneamente residir nos SMs para execução. Este número é crescente entre as gerações de GPUs, permitindo escalabilidade transparente para aplicações. Neste ponto, é importante lembrar que todas as threads de um *warp* executam a mesma instrução. Podemos calcular o número de *warps* que residem em um SM para uma dada configuração de número de blocos e número de threads por bloco definidos para execução do *kernel*. Se cada bloco possui 256 threads então cada bloco terá $256/32 = 8$ *warps*. Suponha que tenhamos 3 blocos por SM, assim temos $8 * 3 = 24$ *warps* por SM.

Este número de *warps* é um importante fator na maneira como CUDA gerencia operações de alta latência, como acesso a memória global. Quando uma instrução executada pelas threads de um *warp* precisa aguardar a resposta de uma operação, o *warp* não é selecionado para execução e outro *warp*, que não está aguardando por resultados é selecionado. Se mais de um *warp* está pronto para execução, então um mecanismo de prioridade é utilizado. Este mecanismo é referenciado como o processo de esconder latência (do inglês *latency hiding*) (Kirk e Hwu [2010], p. 73). Este mecanismo é responsável por manter o *hardware* ocupado o máximo de tempo possível, sendo que, com um maior número de *warps* residentes em um SM, maior é a probabilidade de encontrar um que esteja pronto para execução.

Estes conceitos são importantes no entendimento do funcionamento interno das GPUs e permitem a realização de melhorias significativas no desempenho.

Capítulo 3

Dinâmica de Redes Booleanas Limiarizadas utilizando programação em GPU

Como visto no Capítulo 2, a identificação dos atratores e tamanho das bacias de atração em redes booleanas exige grande esforço computacional, o que impossibilita a manipulação de redes com elevado número de genes. Para amenizar este problema, neste trabalho é proposta a implementação de algoritmos utilizando GPUs.

Neste trabalho utilizamos o modelo de redes booleanas limiarizadas, descrito por Li *et al.* [2004]. A utilização deste modelo é justificada pela representação matricial das funções booleanas, simplificando parte da implementação em GPU. Além disso, a implementação provê uma solução que poderá ser utilizada em outras pesquisas, como por exemplo, na inferência de redes booleanas, realizada nos trabalhos de Higa *et al.* [2011] e Andrade [2012] [1], os quais também utilizam este modelo em seus trabalhos.

O problema a ser resolvido é enunciado como segue:

> Dada uma rede booleana limiarizada $G(V, A)$ composta pelo conjunto V de vértices $V = \{v_1, v_2, ..., v_n\}$, um para cada gene da rede, e uma matriz de regulação gênica $A_{n \times n}$, contendo relações entre os n genes, encontrar quais são os atratores e os respectivos tamanhos das bacias de atração.

Para contagem do tamanho das bacias de atração para redes com até 35 genes (aproximadamente 30 bilhões de estados - 2^{35} estados), foi realizada a análise e simulação completa da dinâmica. Porém, para redes maiores, a tarefa de enumeração de todos os estados torna-se intratável mesmo utilizando GPUs, pois a realização do espaço de estados é astronômica e o tempo de processamento torna-se cada vez mais alto. Neste caso, a contagem do tamanho das bacias de atração é feita de forma estatística, utilizando apenas algumas amostras do espaço de estados. Através deste levantamento estatístico as possíveis bacias de atração com maior volume de estados são identificadas. O mesmo critério é levado em consideração para identificação dos atratores, os quais são parcialmente identificados.

A implementação de Dubrova e Teslenko [2011] mostrou-se promissora (não utiliza GPU) e para utilizá-la com o modelo de redes booleanas limiarizadas, foi implementada uma adaptação da matriz de regulação para respectiva representação em funções booleanas (veja Apêndice A).

[1] Durante este trabalho foi disponibilizada uma versão da implementação em GPU da "Identificação de Atratores utilizando Componentes Conexas em Grafos" para ser utilizada no trabalho de Andrade [2012], trazendo ganhos em desempenho com redução do tempo gasto nas simulações. Alguns resultados obtidos encontram-se na Seção 4.1.

Vale ressaltar que este algoritmo não é uma boa opção para instâncias de redes onde o grau médio de *entrada* dos genes é alto, pois nestes casos o problema também torna-se intratável.

3.1 Metodologia

No desenvolvimento deste trabalho e do software implementado, realizamos diversas atividades, entre as principais estão:

Estudo de Redes Booleanas: Levantamento bibliográfico de trabalhos sobre redes de regulação gênica e redes booleanas. Estudo de técnicas utilizadas para identificação de atratores e bacias de atração;

Estudo de Programação Paralela, GPGPU e CUDA: Levantamento bibliográfico de trabalhos sobre Programação Paralela, GPUs e CUDA. Estudo sobre *hardware* das placas gráficas e seu funcionamento. Estudo de técnicas de programação em GPU utilizando CUDA. Estudo de implementações de algoritmos paralelos para manipulação de grafos. Estudo sobre otimizações de código;

Modelagem e especificação de algoritmos: Especificação e projeto de algoritmos para determinar os atratores e tamanho das respectivas bacias de atração;

Implementação: Implementação em C/C++ e CUDA dos algoritmos especificados;

Validação dos resultados, eficiência e escalabilidade: Testes das implementações, medições de tempo e geração de estatísticas;

Integração com outros softwares: Integração como biblioteca com software de inferência de redes booleanas desenvolvido no trabalho de Tales (Andrade [2012]);

Implementação de aplicativo web: Criação aplicativo para acesso via web do processamento realizado em GPU.

3.2 O Software

Tendo como objetivo a realização da análise da dinâmica de rede booleana limiarizada, para determinar quais são os atratores e tamanho das bacias de atração, utilizando computação de alto desempenho e programação em GPU, este trabalho tem como produto final:

- Uma biblioteca de software;
- Um aplicativo de linha de comandos;
- Um aplicativo web.

A entrada para execução é a descrição da rede booleana limiarizada, representada pela matriz de regulação gênica. Como saída obtém-se a identificação dos atratores e também o tamanho das bacias de atração. Como já visto, descobrir quais são todos os atratores de uma rede booleana é um problema computacionalmente intensivo, o que justifica o uso de programação paralela e de GPUs a fim de obter aumento da capacidade de processamento possibilitando o processamento de um maior número de genes em menor tempo.

Especificamente no modelo de rede booleana limiarizada, dado um estado v_i, para determinar o próximo estado v_j é necessário efetuar uma multiplicação de matriz por um vetor, operação representada pela operação $A\dot{v}_i$ (vide Seção 2.1.3), a qual é intrinsecamente paralela. Além disso, a identificação de um atrator para cada estado é independente da realizada para todos os outros estados, ou seja a realização de trajetórias também independente. Estas características tornam

possível o uso de programação paralela e são as principais motivações para seu uso neste trabalho.

O software utiliza a plataforma CUDA para prover capacidade computacional adicional, promovida pelo processamento em GPU, realizado através de placas gráficas NVIDIA. Os inúmeros processadores da GPU são utilizados para realizar a computação intensiva de identificação dos atratores de forma paralela. Foram implementadas duas estratégias de paralelização. A primeira baseia-se na utilização de um algoritmo de componentes conexas de grafos (veja Seção 3.3.1). Já a segunda, realiza trajetórias em paralelo sobre os estados da rede, até identificar os atratores de cada estado (veja Seção 3.3.2).

Os desempenhos das estratégias foram comparados com relação à duas implementações sequenciais (que não utilizam técnicas de programação paralela). Uma é a implementação sequencial da estratégia básica para a identificação dos atratores (caminhamento para frentes sobre todos os estados). A segunda, a qual despertou grande interesse pelo seu bom desempenho na identificação dos atratores, pois não realiza contagem do tamanho das bacias de atração, é a implementação de Dubrova e Teslenko [2011], a qual utiliza resolvedores SAT, mostrando-se promissora para redes com elevado número de genes e vértices com *baixo grau de entrada* (de 2 a 5). Porém, esta abordagem, torna-se intratável, mesmo para redes pequenas com vértices com alto grau de entrada.

A implementação base dos algoritmos em GPU foi realizada utilizando a linguagem de programação C/C++ e CUDA. Para cada algoritmo foi gerado um módulo executável de linha de comandos. Além disso foi gerada uma biblioteca para integração com outros softwares, por exemplo, software de inferência de redes desenvolvido no trabalho de Andrade [2012]. Ainda, como prova de conceito, também foi implementada uma aplicação web, utilizando Java™, a qual se integra à biblioteca (veja Apêndice B). Essa aplicação foi desenvolvida a fim de disponibilizar o software de uma maneira transparente para usuários que não possuem familiaridade com programação ou ambientes de execução com GPUs. Futuramente a aplicação pode ser expandida, incorporando outras funcionalidades, como a inferência de redes booleanas.

O código fonte do software encontra-se disponível em http://code.google.com/p/cuda-bioinfo-gens.

3.2.1 Arquitetura

O software foi implementado de forma modular, sendo os algoritmos principais como parte de uma biblioteca (*libGeneAnalisysGPU*), a qual possui a implementação em CUDA dos algoritmos especificados na Seção 3.3. Com esta biblioteca, foram criados módulos específicos para execução em *linha de comandos* e também uma *aplicação web*.

Para disponibilização do aplicativo web, a fim de isolar a execução em GPU em uma máquina que não fique necessariamente exposta para web, foi proposta a arquitetura da Figura 3.1. Esta arquitetura possibilita a distribuição da aplicação em duas máquinas. A aplicação executando na primeira máquina responde a requisições HTTP de um navegador web, já a aplicação executando na segunda máquina recebe requisições da primeira para executar os algoritmos em GPU. Desta forma a máquina que possui as GPUs pode ser diferente da máquina que responde as requisições do navegador do usuário, evitando desta forma, sua exposição na web. As aplicações web foram implementadas em Java e executam sobre o servidor web *Apache Tomcat* [2]. A execução dos algoritmos em GPU é realizada através de uma chamada para função nativa (utilizando JNI - *Java Native Interface*), a qual realiza a execução do algoritmo em GPU, utilizando a biblioteca.

Os principais componentes são:

[2] http://tomcat.apache.org/

Biblioteca *libGeneAnalisysGPU*: Biblioteca de software que contém a implementação dos algoritmos utilizando GPUs (C/C++ e CUDA);

CommandLine: Aplicação de linha de comandos utilizada para geração de matriz de regulação gênica aleatória, processamento baseado em grafos utilizando GPU, processamento por Trajetórias em Paralelo utilizando GPU, geração de arquivo no formato de entrada para algoritmo baseado em SAT;

Aplicação web: Dividida em *ServerGeneAnalisys* e *ServerGPUGeneAnalisys*. São responsáveis pela execução do programa pela web. Como entrada recebe um arquivo texto contendo uma matriz de regulação e como saída, devolve os atratores e respectivos tamanhos das bacias de atração.

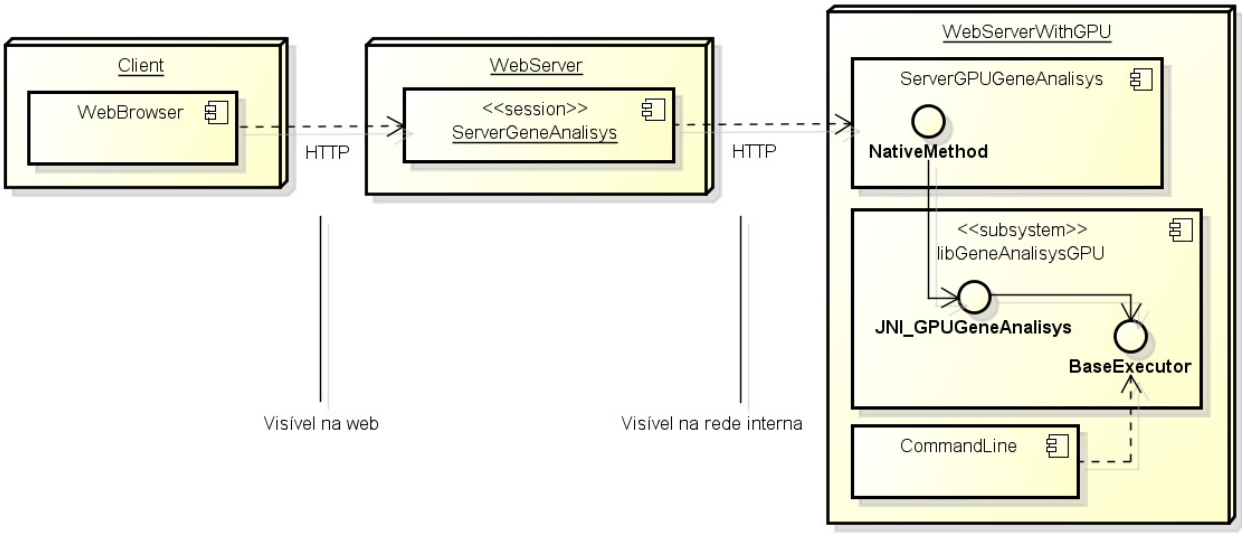


Figura 3.1: *Visão geral da arquitetura proposta para o software. A arquitetura possibilita a separação da aplicação em duas máquinas. A primeira máquina responde a requisições HTTP e é visível na web. Já a segunda, se comunica diretamente com a primeira e executa os algoritmos em GPU, porém é visível somente na rede interna.*

Um breve detalhamento de como estes componentes estão relacionados pode ser visto na Figura 3.2, onde é mostrada uma visão de classes do modelo interno do sistema. Nesta visão, os componentes estão agrupados por módulos.

O módulo ou pacote "command.line.app" contém classes relacionadas à execução por de linha de comandos. Através da execução por linha de comandos é possível executar a geração de matriz de regulação aleatória (*RandomMatrixGenerator*), conversão de arquivo de matriz de regulação para o formato de funções booleanas (*MatrixToFunctionConverter*) e execução dos algoritmos específicos para análise da dinâmica de rede.

O módulo "server.gpu.gene.analisys" contém a aplicação web que executa do lado do servidor (o qual contém as GPUs). Na execução pela web, é este módulo que executa a chamada para interface "nativa" de execução em GPU (interna à biblioteca *libGeneAnalisysGPU*). Os algoritmos possuem uma abstração de execução (*BaseExecutor*) a qual controla informações sobre a execução, como tempo de processamento e o progresso atual. Estas informações são utilizada para informar o usuário a situação atual da execução. Os algoritmos específicos *GraphExecutorGPU* e *ParallelExecutorGPU* implementam a abstração base e executam em GPU. Estes fornecem as informações específicas de sua execução e progresso atual.

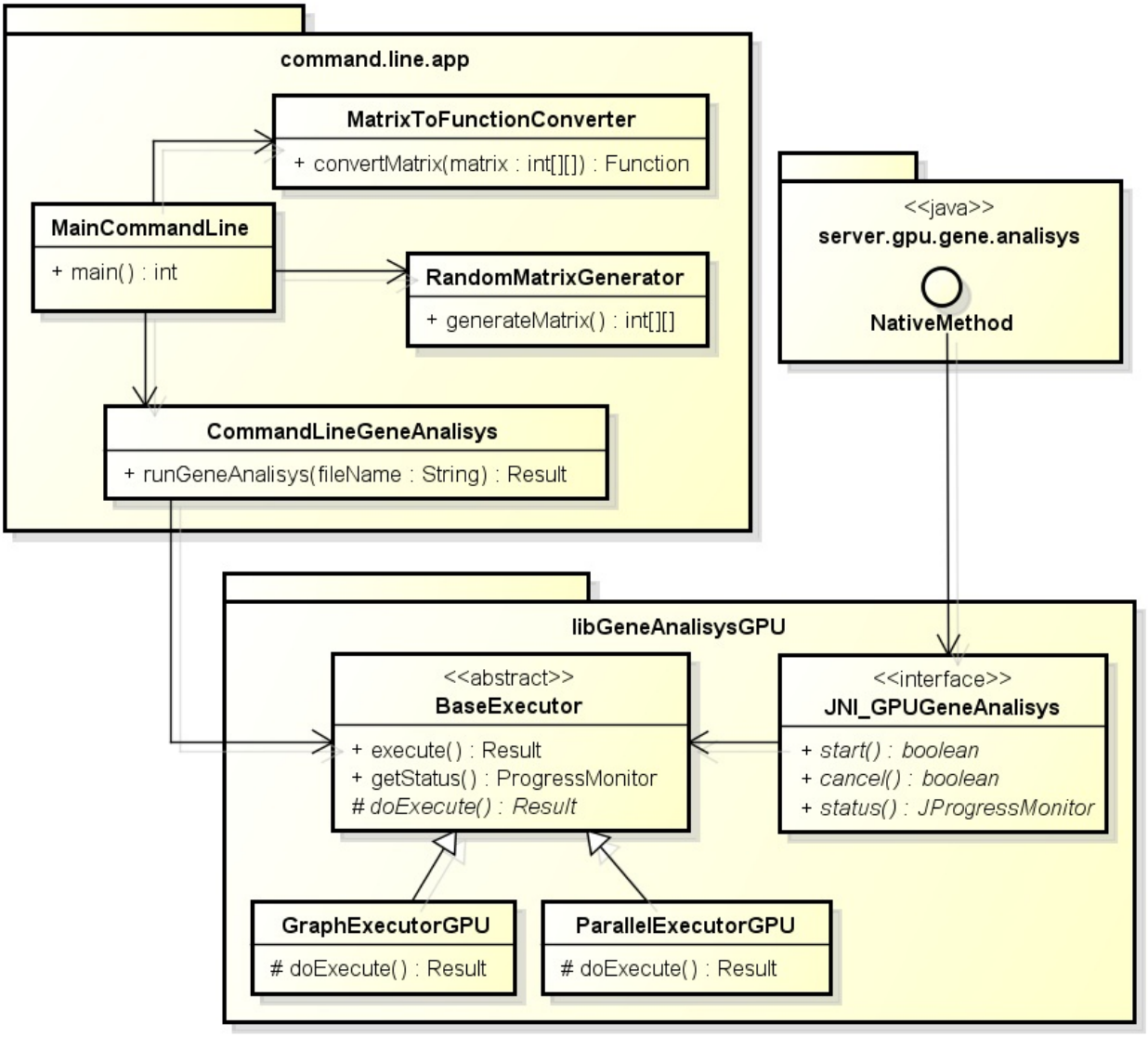


Figura 3.2: *Visão das principais classes do sistema e suas operações. A execução por linha de comandos executa os módulos de geração de matriz de regulação gênica aleatória (*RandomMatrixGenerator*), conversão de matriz de regulação gênica em função booleana (*MatrixToFunctionConverter*) e execução dos algoritmos de* componentes conexas *e de* trajetórias em paralelo. *Da mesma forma, uma chamada nativa provinda da aplicação Java executa algoritmos.*

3.3 Algoritmos

Nesta seção serão apresentados dois algoritmos implementados utilizando GPUs, dos quais um é baseado em *componentes conexas de grafos* e o outro na realização de *trajetórias em paralelo*.

3.3.1 Identificação de Atratores utilizando Componentes Conexas em Grafos

A utilização de algoritmos de componentes conexas em grafos para determinar os atratores em redes booleanas é vista em diversos trabalhos, como Dubrova *et al.* [2005], Skodawessely e Klemm [2011] e Akutsu *et al.* [2008].

De fato, no grafo de transições de estado de uma rede booleana, diferentes bacias de atração correspondem a diferentes componentes conexas e cada componente conexa contém exatamente um ciclo direcionado, o que torna possível a enumeração das bacias de atração de forma direta com relação às componentes conexas. No contexto do algoritmo, o grafo de transições de estado $D(V, E)$,

é um grafo com um conjunto finito não vazio de vértices $V = \{v_0, v_1, \ldots, v_{n-1}\}$ e um conjunto E de arestas ou pares ordenados de vértices dirigidos. Cada aresta (x, y) possui uma única direção de x para y, sendo válidas as seguintes propriedades:

- Grau de entrada de x: número de arestas convergentes a x;
- Grau de saída de x: número de arestas divergentes de x;
- Sumidouro: um vértice com grau de saída nulo;
- Fonte: um vértice com grau de entrada nulo;
- O grau de saída de cada vértice x é 1. Cada vértice possui apenas uma aresta de saída;
- Não existe nenhum vértice caracterizado como sumidouro;
- Cada componente conexa possui um único ciclo.

Estas propriedades são úteis na determinação atrator, uma vez existe uma relação de *um para um*, de um atrator para uma componente conexa. Outra observação interessante é que o modelo para armazenagem dos dados utilizado pode ser simplificado, por exemplo, o vetor de estados E representa diretamente o grafo de transições de estado, apenas com valores decimais, como pode ser visto na Figura 3.3 e na Tabela 3.1.

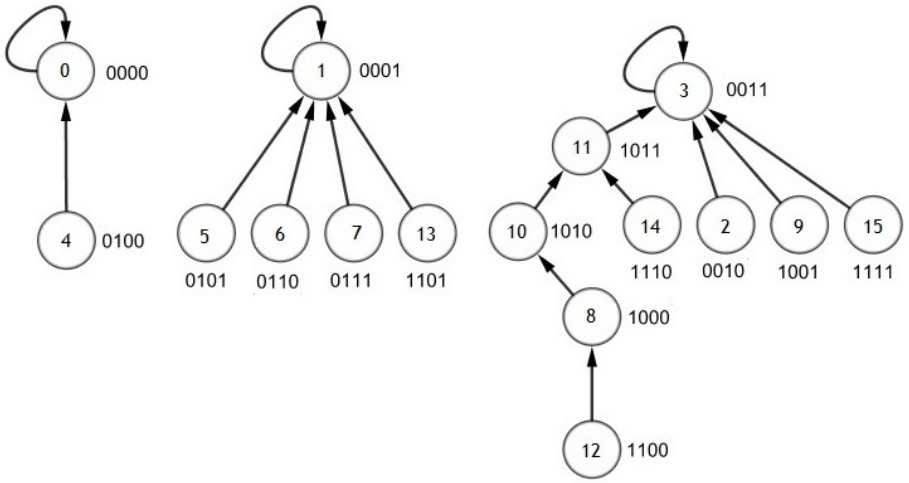


Figura 3.3: *Grafo de transições de estado, com vértices mapeados para valores inteiros.*

E:

Estado (i)	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
Próximo	0	1	3	3	0	1	1	1	10	3	11	3	8	1	11	3

Tabela 3.1: *Grafo de transições de estado na representação decimal. Para cada estado, identificado pela própria posição (i) do vetor E, temos a identificação do próximo estado em forma decimal. Veja que a simples representação decimal pode ser utilizada.*

No vetor E cada posição i, $i = 0, \ldots, (2^n - 1)$, representa um estado da rede no formato decimal, enquanto que o valor "Próximo" no índice especificado representa o próximo estado. Com isso todos os estados são armazenados no formato decimal. Os genes ou os vértices são indexados de $0, \ldots, (n-1)$, assim, cada estado $x = \{v_0, v_1, \ldots, v_{n-1}\}$ (vetor de binário de n bits) é mapeado para o valor numérico decimal inteiro $\mathbf{x}$ o qual representa. Veja que para a representação decimal descrita acima a indexação dos genes foi levemente modificada, passando da sequência $1, \ldots, n$ para $0, \ldots, (n-1)$, o qual será utilizado deste ponto do texto em diante.

Este mapeamento é utilizado para enumerar os vértices e simplificar a manipulação das estruturas de dados. Uma operação importante é saber qual o valor de um gene v_i, dado um estado **x** na representação decimal. Este valor pode ser obtido utilizando a função de mapeamento da Equação 3.1:

$$v_i = (\mathbf{x} \text{ div } 2^{(n-i-1)}) \bmod 2, \tag{3.1}$$

onde n é o número de genes, $i = 0, \ldots, (n-1)$ e *div* e *mod* operadores de divisão e resto, respectivamente, para números inteiros. Em termos de nomenclatura e uso nos algoritmos, a função de mapeamento será nomeada como Π. Sua entrada é um *estado da rede no formato decimal*, e a saída um *vetor binário* correspondente aos estados dos genes. Por exemplo, para o estado $x = 3$, na representação decimal, e considerando 3 genes, temos a seguinte representação binária:

$$v_2 = (3 \text{ div } 2^0) \bmod 2 = 0$$
$$v_1 = (3 \text{ div } 2^1) \bmod 2 = 1$$
$$v_0 = (3 \text{ div } 2^2) \bmod 2 = 1$$

Em resumo temos $\Pi(3) = \{011\}$, ou seja:

Decimal	Gene	v_2	v_1	v_0
3	Binário	0	1	1

Utilizando a abordagem de componentes conexas em grafos, foi realizada a implementação em GPU de um algoritmo para identificar os atratores e os respectivos tamanhos das bacias de atração, tendo como ponto de entrada a matriz de regulação gênica. A primeira fase do algoritmo é relativa a construção do grafo de transições de estado utilizando a matriz de regulação gênica. Em seguida, é realizada a rotulação das componentes conexas, para as quais o atrator é identificado com um simples caminhamento no grafo, até um ciclo ser localizado. Os estados deste ciclo são os estados do atrator. Como cada componente conexa está relacionada a uma bacia de atração o tamanho da bacia é igual ao número de elementos na componente conexa. A complexidade desta abordagem está na implementação de um algoritmo de componentes conexas, que oferece certa liberdade de paralelismo, porém limitado pela quantidade disponível de memória da GPU. Foi utilizado como referência a implementação do algoritmo proposto por Hawick *et al.* [2010], que considera o grafo totalmente carregado em memória.

As etapas para identificação dos atratores e contagem do tamanho das bacias de atração são:

1. Construção do grafo de transições de estado
2. Identificação das componentes conexas
3. Identificação dos atratores
4. Contagem do tamanho das bacias de atração

Cada uma destas etapas será detalhada em seguida.

Construção do grafo de transições de estado

No modelo de rede booleana limiarizada, em uma rede $G(V, A)$, com n genes, conjunto de genes $V = \{v_0, v_1, \ldots, v_{n-1}\}$ e matriz de regulação gênica A, o elemento $v_i(t)$ representa o valor do gene i no tempo t. Uma transição de estado x para y, onde $x = \{x_0, x_1, \ldots, x_{n-1}\}$ com $x_i = v_i(t)$ e $y = \{y_0, y_1, \ldots, y_{n-1}\}$ com $y_i = v_i(t+1)$, é dada pela operação $A \bullet x$ (veja Equação 2.5).

Para construir o grafo de transições de estado basta computar o conjunto de todas as suas transições (ou seja suas arestas) mantendo o mesmo em uma estrutura de dados em memória. Cada aresta para ser computada depende exclusivamente da matriz de regulação e do estado atual da rede. Assim para cada estado da rede, a computação da aresta para o mesmo pode ser realizada em paralelo, uma vez que não existe dependência de nenhuma outra aresta ainda não computada. Nesta fase a computação é intensa, visto que para cada vértice, deve-se computar a multiplicação da matriz de regulação pelo vetor de bits do estado atual.

Construção sequencial

Para construir o grafo sequencialmente, efetuamos para todos os estados $\mathbf{x}$, $\mathbf{x} = 0, \ldots, (2^n - 1)$, no formato decimal, a operação $A \bullet \Pi(\mathbf{x})$. O resultado é armazenado em um vetor E de arestas, o qual armazena a representação das transições de estado da rede booleana. O índice k neste vetor representa o estado $\mathbf{x}$ ($\mathbf{x} = k$) e o seu valor y ($\mathbf{y} = E[k]$), indica que existe uma transição de estado $\mathbf{x} \rightarrow \mathbf{y}$ no formato decimal, valendo o mesmo no formato binário, ou seja $\Pi(\mathbf{x}) \rightarrow \Pi(\mathbf{y})$. O Algoritmo 1 abaixo representa esta construção:

Algoritmo 1: Construção sequencial do grafo de transições de estados

Entrada: A (matriz de regulação gênica n x n), n (número de genes em questão).
Saída: E vetor de arestas. Aresta $\mathbf{x}$ para $\mathbf{y}$ é dada por $\mathbf{y} = E[\mathbf{x}]$.

```
1  para x de 0 até (2^n − 1) faça
2      E[x] ← A • Π(x) ;
3  fim
4  return E ;
```

Uma abordagem direta para realizar a paralelização da construção do grafo seria a execução da linha 2 do algoritmo em paralelo, abordagem que foi implementada também em GPU. Basicamente foi implementado um *kernel* que contém a linha 2 do algoritmo, realizando endereçamento das posições de E pelo identificador das threads. Além disso, operação da linha 2 ($A \bullet \Pi(\mathbf{x})$) é da ordem de $O(n^2)$ pois cada elemento (i, j) da matriz A precisa ser multiplicado pelo respectivo elemento (j) do estado $\Pi\mathbf{x}$. Este tempo pode ser melhorado utilizando uma sequência de estados $v_0 \ldots v_{(k-1)}$ específica, a qual produza estados que possuam apenas um único bit de diferença, com isso valores já computados podem ser reutilizados. Esta sequência é possível de ser gerada utilizando o *Código Gray*.

Construção utilizando *Código Gray*

O *Código Gray* é uma codificação para sequências binárias, onde cada elemento da sequência é diferente do seguinte por apenas um único bit (Doran [2007], p. 1573). Foi inventado por Frank Gray e descrito formalmente em uma patente no ano de 1953 (Gray [1953]). No contexto deste trabalho desejamos uma função na qual dado um valor inteiro decimal, obtemos como resultado o respectivo valor em decimal na codificação *gray*. Desta forma, para um dado inteiro decimal, o valor *gray* decimal é obtido utilizando-se a Equação 3.2 (Doran [2007], p. 1581).

$$v_{gray} = v \oplus (v \text{ div } 2), \tag{3.2}$$

onde, $\oplus$ é a operação de *ou exclusivo* (XOR) e *div* é a operação de divisão para números inteiros.

Um exemplo pode ser visto na Tabela 3.2, na qual temos os estados de 0 à 15 nas representações decimal, binário, codificação *gray decimal* e codificação *gray binário*.

Decimal	Binário	Gray Decimal	Gray Binário
0	0000	0	0000
1	0001	1	0001
2	0010	3	0011
3	0011	2	0010
4	0100	6	0110
5	0101	7	0111
6	0110	5	0101
7	0111	4	0100
8	1000	12	1100
9	1001	13	1101
10	1010	15	1111
11	1011	14	1110
12	1100	10	1010
13	1101	11	1011
14	1110	9	1001
15	1111	8	1000

Tabela 3.2: *Exemplo de codificação* gray *para sequência numérica de* 0 *a* 15.

Na operação $A \bullet \Pi(\mathbf{x})$, a matriz A é multiplicada pelo vetor binário $\Pi(\mathbf{x})$, sendo que em seguida o resultado desta multiplicação é limiarizado. Para diminuir o número de operações aritméticas necessárias para computar essa operação (utilizando a multiplicação padrão temos $O(n^2)$ operações aritméticas), podemos utilizar a sequência gerada pelo código *gray*, fazendo a construção do grafo seguir sua ordem. Para isso, é armazenado o valor de *entrada* (valor da multiplicação de matriz - antes da limiarização) para do estado anterior na sequência de geração de estados. Com isso, para multiplicação de matriz para o elemento atual da sequência, utiliza-se a entrada pré-calculada, sendo necessário apenas atualizar a entrada, adicionando ou removendo, valores da matriz A com relação à coluna de mesmo valor de índice onde houve a mudança de bit.

Suponha qualquer dois estados $s_1 = (v_0, v_1, \ldots, v_{(n-1)})$ e $r_1 = (u_0, u_1, \ldots, u_{n-1})$ da rede, para os quais a diferença seja de um único bit, de índice j, e assim $v_j \neq u_j$. Com o próximo estado, a partir de s_1 já calculado, sendo o mesmo nomeado por s_2, sua *entrada* (de s_2) é dada pelo vetor I, calculado pela multiplicação da matriz $A \times s_1$ (antes realizar a limiarização) como:

$$I = A \times s_1 = \begin{pmatrix} a_{00} & a_{10} & a_{20} & \ldots & a_{(n-1)0} \\ a_{01} & a_{11} & a_{21} & \ldots & a_{(n-1)1} \\ a_{02} & a_{12} & a_{22} & \ldots & a_{(n-1)2} \\ \ldots & \ldots & \ldots & \ldots & \ldots \\ a_{0(n-1)} & a_{1(n-1)} & a_{2(n-1)} & \ldots & a_{(n-1)(n-1)} \end{pmatrix} \times \begin{pmatrix} v_0 \\ v_1 \\ v_2 \\ \ldots \\ v_{(n-1)} \end{pmatrix} = \begin{pmatrix} p_0 \\ p_1 \\ p_2 \\ \ldots \\ p_{(n-1)} \end{pmatrix},$$

onde, $\times$ é operação usual de multiplicação de matriz e $p_j = \sum_{i=0}^{n-1} a_{ij} v_i$, $j = 0, \ldots, (n-1)$.

Queremos obter o próximo estado r_2 a partir de r_1. Como r_1 difere de s_1 por um único bit j e $s_1 \rightarrow s_2$, podemos utilizar a mesma entrada I de s_2 para o cálculo, reconsiderando apenas os elementos da coluna de índice j da matriz A para atualizar o novo valor de I, com o sinal destes

elementos segundo a diferença de s_1 e r_1, isto é u_j. Se $u_j = 1$, quer dizer que o bit j mudou de 0 para 1, sendo assim os valores da coluna j da matriz A devem considerados em I, logo atualiza-se o vetor I somando-o com o vetor coluna j de A, isto é $(a_{j0}, a_{j1}, a_{j2}, \ldots, a_{j(n-1)})$. Já se $u_j = 0$, quer dizer que os valores do vetor coluna j de A devem ser desconsiderados da entrada, assim subtraímos os mesmos entrada I. Após a atualização da entrada, o valor do próximo estado pode ser computado normalmente, utilizando a limiarização. Com isso, podemos calcular a entrada de r_2 como:

$$I^{r_2} = I + kA_j = \begin{pmatrix} p_0 \\ p_1 \\ p_2 \\ \ldots \\ p_{(n-1)} \end{pmatrix} + \begin{pmatrix} k * a_{j0} \\ k * a_{j1} \\ k * a_{j2} \\ \ldots \\ k * a_{j(n-1)} \end{pmatrix},$$

sendo k constante, dado por:

$$k = \begin{cases} +1 & \text{se} \quad u_j = 1 \\ -1 & \text{se} \quad u_j = 0, \end{cases}$$

onde j é o índice do bit da diferença entre s_1 e r_1. Com isso, o valor do próximo estado r_2, a partir de $r_1 = (u_0, u_1, \ldots, u_{n-1})$, é dado pela limiarização de I^{r_2}, ou seja, seguindo a sequência *gray*, como cada elemento difere do próximo por apenas um bit, para calcular o próximo estado de cada elemento, é necessário apenas calcular a nova entrada, considerando a entrada anterior e a coluna da matriz de regulação com respeito à diferença binária. Tanto a soma quanto a determinação de k são realizadas em tempo linear ou constante, ou seja todas as multiplicações de matriz ocorrerão na ordem de $O(n)$ operações.

Por exemplo, considere a matriz de regulação gênica

$$T = \begin{array}{c} \\ v_0 \\ v_1 \\ v_2 \\ v_3 \end{array} \begin{array}{c} \begin{array}{cccc} v_0 & v_1 & v_2 & v_3 \end{array} \\ \begin{pmatrix} 0 & 0 & 0 & -1 \\ 0 & -1 & 0 & 0 \\ 1 & -1 & 0 & 0 \\ 0 & 0 & 1 & 0 \end{pmatrix} \end{array}$$

Segundo a sequência gray da Tabela 3.2, iniciamos a obtenção das transições de estados pelo estado $x_1 = \{0000\}$, o qual pela definição da relação de transição de estados leva a $x_2 = \{0000\}$, pois:

$$x_2 = T \times x_1 = \overset{T}{\begin{pmatrix} 0 & 0 & 0 & -1 \\ 0 & -1 & 0 & 0 \\ 1 & -1 & 0 & 0 \\ 0 & 0 & 1 & 0 \end{pmatrix}} \times \overset{x_1}{\begin{pmatrix} 0 \\ 0 \\ 0 \\ 0 \end{pmatrix}} = \overset{I}{\begin{pmatrix} 0 \\ 0 \\ 0 \\ 0 \end{pmatrix}} \cong \overset{x_2}{\begin{pmatrix} 0 \\ 0 \\ 0 \\ 0 \end{pmatrix}}$$

Após esta operação, temos que $I = \{0, 0, 0, 0\}$. O próximo estado ter sua transição determinada é $y_1 = \{0001\}$, o qual leva a y_2. É fácil verificar que $x_1 = \{0000\}$ e y_1 diferem de apenas um bit, no índice 3, com o mesmo passando de 0 para 1, portanto $k = 1$. A transição de y_1 para y_2 será calculada utilizando apenas o valor atual de $I = \{0, 0, 0, 0\}$ somado com o vetor coluna 3 de T, dado por $T_3 = \{-1, 0, 0, 0\}$. Desta forma:

$$y_2 = I + kT_3 = \begin{pmatrix} 0 \\ 0 \\ 0 \\ 0 \end{pmatrix} + \begin{pmatrix} 1*-1 \\ 1*0 \\ 1*0 \\ 1*0 \end{pmatrix} = \overset{I}{\begin{pmatrix} -1 \\ 0 \\ 0 \\ \mathbf{0} \end{pmatrix}} \cong \overset{y_2}{\begin{pmatrix} 0 \\ 0 \\ 0 \\ \mathbf{1} \end{pmatrix}}$$

Veja que a operação também atualiza o valor de I. Como foi aplicada a limiarização, foi mantido o valor 1 para o gene v_3 no estado y_2 pois $I_3 = 0$. Para o próximo estado da sequência gray $z_1 = \{00\mathbf{1}1\}$ é feita a mesma operação. É fácil verificar que $y_1 = \{0001\}$ e z_1 diferem de apenas um bit, no índice 2, com o mesmo passando de 0 para 1, portanto com $k = 1$, $I = \{-1, 0, 0, 0\}$ e $T_2 = \{0, 0, 0, 1\}$, fazemos:

$$z_2 = I + kT_2 = \begin{pmatrix} -1 \\ 0 \\ 0 \\ 0 \end{pmatrix} + \begin{pmatrix} 1*0 \\ 1*0 \\ 1*0 \\ 1*1 \end{pmatrix} = \overset{I}{\begin{pmatrix} -1 \\ 0 \\ 0 \\ 1 \end{pmatrix}} \cong \overset{z_2}{\begin{pmatrix} 0 \\ 0 \\ \mathbf{1} \\ 1 \end{pmatrix}}$$

Como anteriormente, a operação também atualiza o valor de I. Da mesma forma foi aplicada a limiarização e mantido o valor 1 para o gene v_2 no estado z_2, pos $I_2 = 0$. Para o próximo estado da sequência gray $w_1 = \{001\mathbf{0}\}$ é feita a mesma operação. Já que $z_1 = \{0011\}$ e w_1 diferem de apenas um bit, no índice 3, com o mesmo passando de 1 para 0, portanto, desta vez $k = -1$, $I = \{-1, 0, 0, 1\}$ e $T_3 = \{-1, 0, 0, 0\}$, fazemos:

$$w_2 = I + kT_3 = \begin{pmatrix} -1 \\ 0 \\ 0 \\ 1 \end{pmatrix} + \begin{pmatrix} -1*-1 \\ -1*0 \\ -1*0 \\ -1*0 \end{pmatrix} = \overset{I}{\begin{pmatrix} 0 \\ 0 \\ 0 \\ 1 \end{pmatrix}} \cong \overset{w_2}{\begin{pmatrix} 0 \\ 0 \\ 1 \\ \mathbf{1} \end{pmatrix}}$$

Com isso temos transições mostradas na Tabela 3.3, as quais são as mesmas da Figura 2.5, mostrada novamente na Figura 3.4, com destaque para os estados calculados.

Estado		**Próximo**	
x_1	0000	x_2	0000
y_1	0001	y_2	0000
z_1	0011	x_2	0011
w_1	0010	y_2	0011

Tabela 3.3: *Resumo dos estados calculados utilizado sequência gray.*

Para todos os outros estados da sequência da Tabela 3.3, pode ser feito o mesmo procedimento, gerando o grafo de transições de estado completo.

A abordagem utilizando a sequência *gray* não foi utilizada efetivamente em nenhuma implementação em GPU, pois a computação dos estados dependem totalmente da sequência e de estados pré-computados, ou seja não é passível de ser paralelizado. A mesma foi apresentada neste trabalho pois mostra um ganho interessante na redução da complexidade na multiplicação de matrizes com as características apresentadas, podendo ser utilizada futuramente em outros trabalhos.

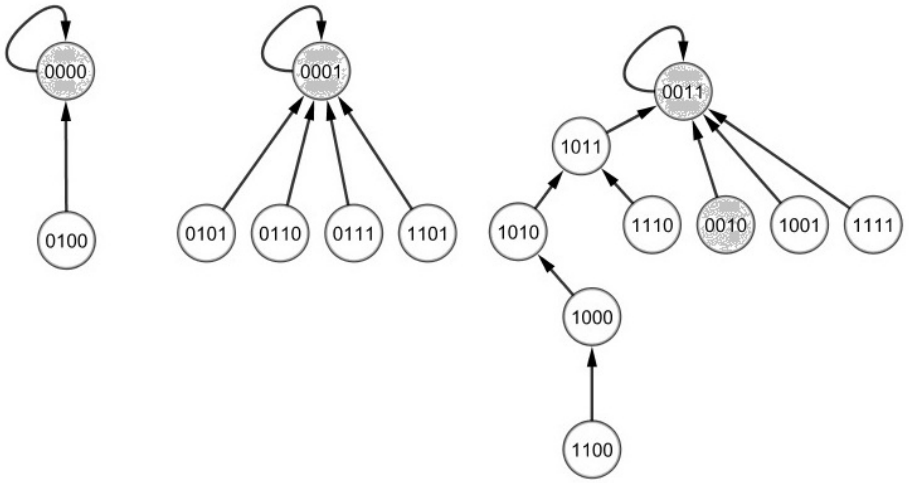


Figura 3.4: *Diagrama de sequência de estados gerado pela sequência de gray. Os estados calculados pelo exemplo foram* {0000}, {0001}, {0011} *e* {0010}.

Identificação das Componentes Conexas

Nesta etapa do algoritmo, foi utilizada uma versão da implementação em GPU proposta por Hawick *et al.* [2010]. Este algoritmo efetua a identificação das componentes do grafo de transições de estado utilizando um *rótulo de identificação* para cada vértice. Ao final, vértices com o mesmo identificador pertencem à mesma componente conexa, logo pertencem à mesma bacia de atração. A identificação de um vértice é um valor inteiro decimal, que representa valor dos genes em um dado instante de tempo, obtidos através do mapeamento decimal/binário (Equação 3.1). O algoritmo compara identificadores de pares de vértices, rotulando os vértices com o valor do *menor identificador*, o qual é propagado para os próximos vértices, caso também ganhe na comparação. Este processo é feito em paralelo, através de várias iterações. As comparações entre os pares de vértices pode ser feita em paralelo pois cada comparação é independente uma da outra. O fim da rotulação ocorre quando os valores estabilizam, isto é, quando todos os vértices possuem valor único de rótulo, sendo assim, não é mais possível realizar nenhuma alteração entre os rótulos. Com isso, ao final os vértices da mesma componente conexa estarão rotulados com o mesmo identificador. Uma ilustração geral do processo pode ser vista nas figuras 3.5 e 3.6.

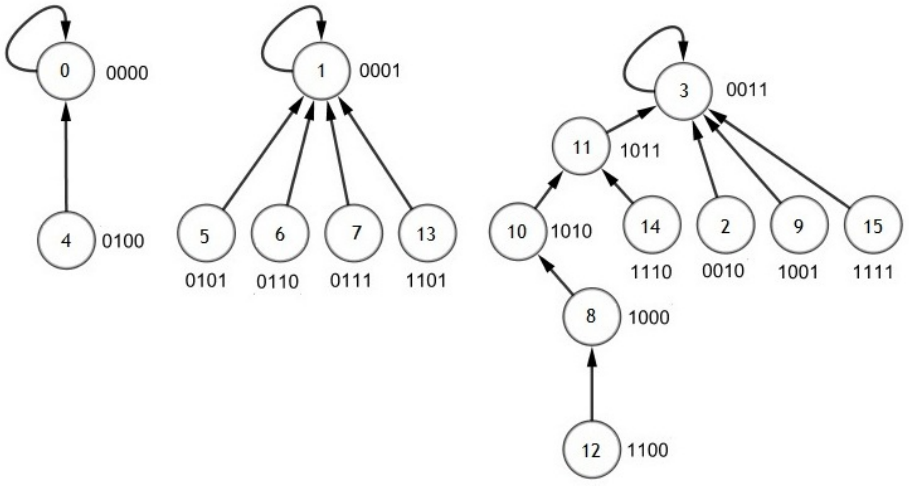


Figura 3.5: *Grafo de sequência de estados para identificação das componentes conexas. Contém a representação decimal e binária de cada vértice.*

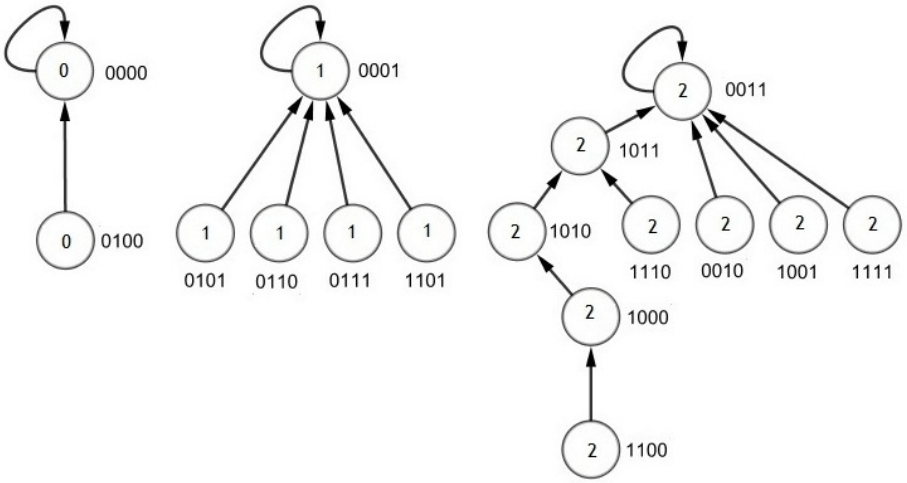


Figura 3.6: *Grafo de sequência de estados com as componentes conexas identificadas. Cada componente conexa foi identificada com o valor do menor vértice da componente, assim todos os vértices da componente foram rotulados com este valor. A componente de rótulo* 0 *possui dois elementos:* 4 = {0100} *e* 0 = {0000}*, o que significa que sua bacia de atração tem tamanho* 2*. As outras componentes possuem rótulos* 1 *e* 2*, sendo de tamanhos* 5 *e* 9*, respectivamente.*

Os procedimentos para execução no computador hospedeiro e em GPU são vistos nos Algoritmos 2 e 3, respectivamente. O Algoritmo 2 realiza alocação para memória global necessária na GPU. Também define os parâmetros para execução do *kernel*, descrito pelo Algoritmo 3, e efetua a verificação da condição de parada. A execução do algoritmo Algoritmo 2 ocorre após a criação do grafo de transições de estado, o qual é armazenado em um vetor E de arestas. Cada posição $j = E[i]$, $i = 0 \ldots 2^n$, refere-se a uma aresta do estado s_i para s_j, onde i é o índice do vetor e j o valor de $E[i]$, como visto anteriormente na Tabela 3.1. O valor do rótulo das componentes é armazenado em um vetor saída C. O valor de $c = C[i]$ representa a componente c do vértice i. Inicialmente $C[i] = i$, para todo i (linhas 1 a 4).

Nas linhas 5 a 11, é feita transferência dos dados para memória global da GPU, sendo os vetores E e C identificados na GPU por Ed e Cd, respectivamente (o sufixo d é para identificar uma variável na memória da GPU - d : *device*). O *kernel* pode ser executado várias vezes, até que os vértices da mesma componente, possuam o mesmo identificador (laço das linhas 12 a 16). Para identificar a condição de parada é utilizada a variável m, a qual armazena se houve alguma modificação de valores em C. Na memória da GPU m é identificada como md.

A cada iteração do laço das linhas 12 a 16 o *kernel* representado pelo Algoritmo 3 é executado. Nele, cada thread efetua uma comparação entre os rótulos de um par de vértices (linhas 5 a 11) e se necessário, o identificador do menor vértice é propagado como identificador da componente para o vértice que perdeu a comparação. Havendo a propagação, a variável md possui seu valor alterado para `true`, indicando que houve propagação. Após a comparação de todos os pares de vértices, o valor de md é copiado para a memória do computador hospedeiro na variável m. O laço é executado até que não hajam propagações de rótulos, sendo assim, até que m tenha valor `false`. Esta condição de parada indica que todas as componentes conexas foram definidas, sendo que os rótulos $Cd[i]$, possuem a identificação de cada componente como o menor valor entre os identificadores de vértices da componente na qual se encontram. O vetor Cd é então copiado para memória do computador hospedeiro em C, o qual é retornado pelo algoritmo. Após isso será feito um pós processamento sobre C para identificar os atratores e tamanho das bacias de atração.

Algoritmo 2: Identificar Componentes Conexas (Hospedeiro)

Entrada: E (arestas do grafo na memória do computador hospedeiro), n (número de vértices), K (numero de threads para execução do kernel)
Saída: C (identificação das componentes conexas para cada vértice atualizada).

```
1  // Inicializa as componentes como C[i] = i
2  for i = 0 to 2^n − 1 do
3  |        C[i] ← i ;
4  end
5  alocarEmGPU(Ed, |E|) ; // Alocação de memória em GPU para Ed
6  alocarEmGPU(Cd, |E|) ; // Alocação de memória em GPU para Cd
7  alocarEmGPU(md, 1) ; // Alocação de memória em GPU para md
8  // Copia dos dados para memória global da GPU
9  copiarParaGPU(E, Ed) ;
10 copiarParaGPU(C, Cd) ;
11 repeat
12 |        copiarParaGPU(md, false) ;
13 |        «executar em paralelo na GPU usando |K| threads»
   |        Kernel Identificar Componentes Conexas(Ed, Cd, md) ;
14 |        // Atualiza na memória do host o valor da flag que indica a
   |           finalização do algoritmo
15 |        copiarParaHost(md, m) ;
16 until m = false;
17 copiarParaHost(Cd, C) ;
18 desalocarDaGPU(Cd) ;
19 desalocarDaGPU(Ed) ;
20 desalocarDaGPU(md) ;
21 return C ;
```

Algoritmo 3: Kernel Identificar Componentes Conexas (GPU)

Entrada: Ed (arestas do grafo na memória da GPU), Cd (identificação das componentes conexas de cada vértice), md (armazena, na memória da GPU, se houve propagação do rótulo de algum vértice).
Saída: Cd identificação das componentes conexas atualizada, md (armazena, na memória da GPU, se houve propagação do rótulo de algum vértice).

```
1  i ← threadID ; // Vértice i, obtido do ambiente CUDA
2  j ← Ed[i] ;
3  c_i ← Cd[i] ;
4  c_j ← Cd[j] ;
5  se c_i < c_j então
6  |        Cd[j] ← c_i ;
7  |        md ← true ;
8  fim
9  senão se c_j < c_i então
10 |        Cd[i] ← c_j ;
11 |        md ← true ;
12 fim
```

Podemos ver a evolução da identificação das componentes conexas na Figura 3.7, onde as

imagens (*a*), (*b*), (*c*) e (*d*) mostram a propagação do valor do menor vértice, até ser propagado entre todos os vértices de cada componente, identificando a mesma de forma única. A ordem da propagação nem sempre é a mesma, pois varia dependendo da ordem da execução das threads da GPU. Ao final a identificação de cada componente é utilizada para determinação dos atratores, pois cada componente identifica uma bacia de atração.

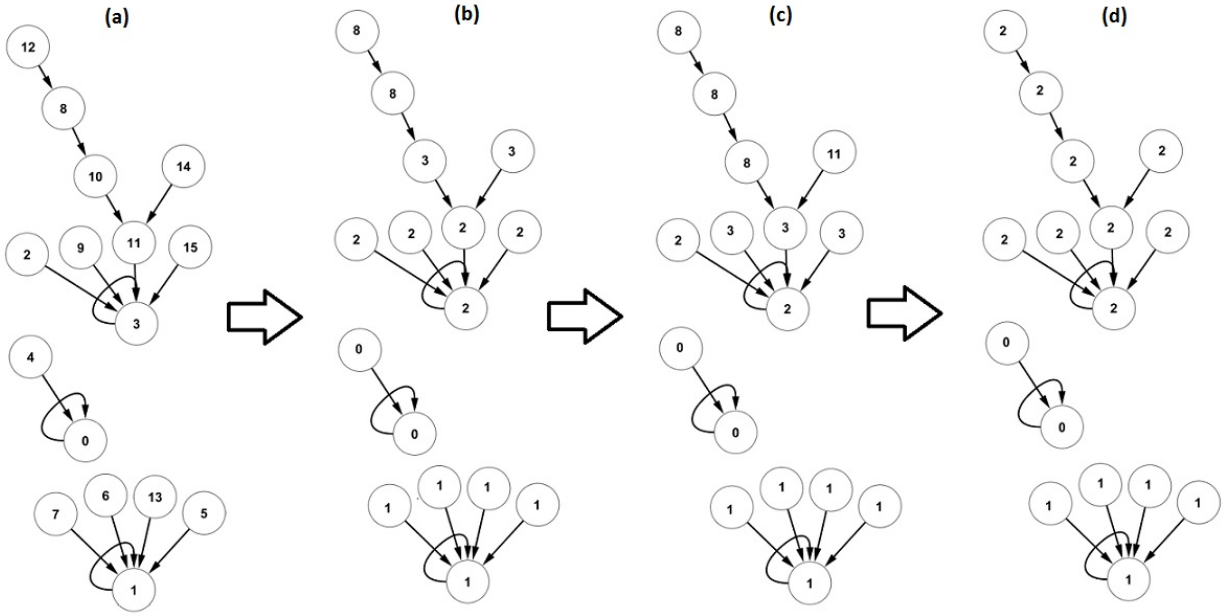


Figura 3.7: *Passos da execução do Algoritmo 2. Na figura os vértices armazenam a identificação da componente. (a) Inicialmente* $C[i] = i$*, para todo* i*,* $i = 1 \ldots 2^n$*. (b) (c) e (d) são três iterações referentes a execução do kernel nas linhas 11 a 16 do Algoritmo 2. Cada iteração, executa o Algoritmo 3, o qual propaga o menor identificador de cada componente conexa. A propagação é feita em paralelo sobre os pares de vértices.*

Determinação dos atratores e Contagem do tamanho das bacias de atração

O último passo do algoritmo é a identificação dos atratores e contagem do tamanho das bacias de atração a partir das componentes conexas calculadas. Estes dois procedimentos são realizados no Algoritmo 4 que recebe como entrada o grafo de transições de estado E e a identificação das componentes conexas C, fornecendo como saída, uma tabela de espalhamento *hash* (chave/valor) T, onde para cada elemento temos:

- *chave:* conjunto de estados do atrator (identificadores no formato decimal);
- *valor:* tamanho da bacia de atração do atrator armazenado na chave.

O Algoritmo 4 primeiramente efetua a contagem do tamanho das bacias de atração, realizando a contagem do número de estados de cada componente conexa do vetor C, mostrado no laço das linhas 2 a 10. Para isto, o identificador da componente de cada vértice é agrupado no mapa T da seguinte forma: para cada elemento c contido em C, é verificado se c já foi adicionado ao mapa T. Em caso negativo, ou seja, a componente conexa não foi processada ainda, c é inserido em T com valor 1 (uma inserção na tabela para determinada chave remove valores já existentes para a chave, se houver). Caso contrário, o valor contido na posição de c de T ($T.valor(c)$) é incrementado em uma unidade (linha 5 e 6). Após todas as componentes serem processadas a tabela T conterá o tamanho das bacias de atração, porém com as chaves como sendo os identificadores dos rótulos das componentes. Deste ponto em diante, os atratores são determinados realizando-se uma trajetória simples a partir dos estados que estão como chave em T. Para realizar a trajetória é realizado um caminhamento para frente, sobre cada um destes estados. Esse processamento é feito no Algoritmo 5, que recebe como entrada um estado s e o vetor de sequência de estados E e devolve

como saída, o conjunto de estados do atrator ao qual s pertence. Durante o caminhamento é utilizada uma estrutura de dados de pilha (P) para armazenar o caminho realizado. É realizada uma trajetória a partir de s (utilizando E) onde cada estado obtido na trajetória é adicionado à pilha ($P.push(s)$), o que é visto na linha 6. A trajetória termina quando o estado obtido já foi adicionado na pilha. Após isso, o atrator é composto dos estados do topo da pilha até a posição do estado identificado como já adicionado (linha 10 a 13).

Algoritmo 4: Pos-Processamento Componentes Conexas

Entrada: E (arestas do grafo na memória do computador hospedeiro), C (identificador das componentes conexas)
Saída: T (Tabela de espalhamento hash que contém como chave atrator e como valor o tamanho da bacia de atração).

```
1  // Contabiliza o número de elementos em cada atrator
2  for i = 0 to |C| do
3      c ← C[i] ;
4      se T.contem(c) então
5          basinSize ← T.valor(c) + 1 ;
6          T.inserir(c, basinSize) ;
7      fim
8      senão
9          T.inserir(C[i], 1) ;
10     fim
11 end
12 // Troca o rótulo correspondente pelos estados do atrator
13 foreach c in T.chaves() do
14     attractor ← TrajetoriaAteAtrator(c, E) ;
15     T.inserir(attractor, T.get(c)) ;
16     T.remover(c) ;
17 end
18 return T ;
```

Algoritmo 5: Trajetoria Ate Atrator

Entrada: s (estado da rede), E (arestas do grafo)
Saída: $attr$ (Conjunto que contém os estados do atrator obtido a partir do estado s informado)

```
1  // Conjunto para armazenar os estados do atrator
2  attrSet ← initSet() ;
3  // Pilha para armazenar a trajetória realizada
4  P ← initStack() ;
5  repeat
6      P.push(s) ;
7      s ← E[s] ;
8  until P.contem(s);
9  c ← s ;
10 repeat
11     s ← P.pop() ;
12     attrSet.add(s) ;
13 until s = c;
14 return attrSet ;
```

Por exemplo, suponha o grafo de sequência de estados dado pela Figura 3.8, que contém os estados do vetor E. Para este grafo, temos as componentes conexas C da Tabela 3.4.

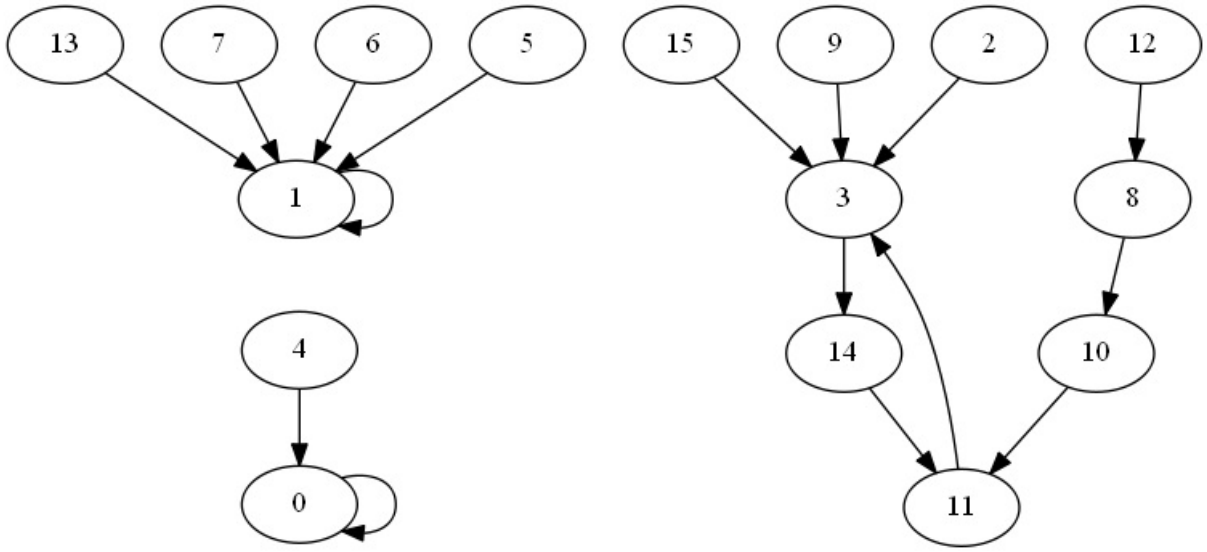


Figura 3.8: *Grafo de sequência de estados para exemplo de contagem do tamanho das bacias de atração. Veja que existe um atrator com 3 estados: {3, 11, 14}*

i	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
E	0	1	3	14	0	1	1	1	10	3	11	3	8	1	11	3
C	0	1	2	2	0	1	1	1	2	2	2	2	2	1	2	2

Tabela 3.4: *Exemplo da identificação de atratores e contagem do tamanho das bacias de atração.*

Na Tabela 3.4, $E[i] = k$ indica que o vértice i possui uma aresta para o vértice k. Assim para o vértice 3 temos que $E[3] = 14$, indicando que o vértice 3 possui uma aresta para o vértice 14. Cada posição do vetor C indica o rótulo da componente conexa para o vértice i. Desta forma, $C[3] = 2$, indica que o vértice 3 é da componente conexa de rótulo 2. Com isso, para contagem do número de componentes basta contabilizarmos o número de repetições de cada rótulo em C, como mostrado na Tabela 3.5.

Rótulo	**Tamanho da Componente**
0	2
1	5
2	9

(a)

Rótulo	**Tamanho da Componente**
{0}	2
{1}	5
{3, 11, 14}	9

(b)

Tabela 3.5: *Tabela (a) contém o tamanho das bacias de atração utilizando os rótulos das componentes conexas. Tabela (b) contém o tamanho das bacias de atração pelos estados do respectivo atrator.*

Na Tabela 3.5 (a), temos que a componente de rótulo 0 possui dois elementos, a de rótulo 1 possui 5 elementos e a de rótulo 2 possui 9 elementos. Esta tabela é construída nas linhas 2 a 10 do Algoritmo 4. Nas linhas 13 a 16, o atrator de cada componente é identificado utilizando, para realização da trajetória, os valores de rótulo da tabela (a) como estados iniciais. Ao final os valores dos rótulos são substituídos pelos estados do atrator correspondente, assim ficamos com a Tabela 3.5 (b) como resultado final.

3.3.2 Identificação de Atratores por Trajetórias em Paralelo em GPU

A implementação do algoritmo baseado em componentes conexas (Seção 3.3.1), mostra bons resultados e tempos de processamento, porém este algoritmo é limitado pela quantidade de memória (RAM) disponível, pois o grafo de transições de estado deve estar completamente carregado em memória. Esta limitação motivou a implementação desta segunda abordagem.

Ideia Básica

O algoritmo baseia-se no fato de que para *um dado estado* a trajetória realizada para identificação de seu atrator é independente da trajetória realizada para identificação dos atratores de todos os outros estados. Com isto, a identificação do atrator à partir de qualquer estado pode ser realizada de forma paralela, por sua trajetória ser independente. Assim, cada estado, no grafo de transições de estado, pode ser processado por uma thread diferente, a qual efetua a trajetória até o atrator. Para identificação única de cada atrator, é utilizado o estado que representa o menor valor decimal entre os estados do atrator. Uma vantagem nesta abordagem é que não é necessário armazenar o grafo de transições de estado em memória e uma desvantagem é que caminhos que levam ao mesmo atrator podem ser visitados mais de uma vez.

Trajetórias em Paralelo utilizando GPU

Para realizar as trajetórias em paralelo utilizando as inúmeras threads da GPU, foi criado o conceito de espaço de trabalho. Um espaço de trabalho consiste em um número fixo de estados que serão processados na execução do kernel. O tamanho do espaço de trabalho é limitado pela quantidade de memória da GPU, onde é levado em consideração o total de memória necessário para execução do kernel, isto é, memória para entrada e saída. Desta forma, o tamanho do espaço de trabalho depende da quantidade de memória global disponível. Cada espaço de trabalho é armazenado em um vetor e é transferido para memória global da GPU. Durante a execução do kernel, cada thread seleciona um estado não processado neste vetor e realiza um caminhamento para frente até identificar seu atrator, o que indica que o processamento do estado selecionado chegou ao fim. O estado selecionado tem então seu atrator identificado (armazenado) no vetor saída. A thread verifica se existem estados a serem processados e seleciona um novo estado, repetindo o processo novamente, até que todos os estados sejam processados.

Todas as threads realizam este comportamento, sendo que cada uma sabe a posição exata de endereçamento do próximo estado disponível a ser processado (cada thread processa um número fixo de estados dentro de um espaço de trabalho). Como a identificação do atrator para cada estado é armazenada na memória global da GPU em outro vetor (saída), ao finalizar o processamento de um espaço de trabalho, os atratores identificados são transferidos para o computador hospedeiro e são armazenados na memória RAM (no caso da implementação em questão são armazenados em uma estrutura de dados de tabela de espalhamento *hash*). Neste momento também é realizada a atualização do tamanho das bacias de atração para cada atrator identificado. Os espaços de trabalho são processados até que todos os estados da rede tenham seus atratores identificados. É importante notar que em nenhum momento todos os estados da rede estarão armazenados, apenas uma parte dos mesmos. Também para a realização da saída, somente os atratores identificados são armazenados.

A programação em GPUs permite a utilização de múltiplas GPUs de forma praticamente transparente para a aplicação. Porém, é necessário realizar um novo nível de abstração de paralelismo. A divisão em espaços de trabalho permite facilmente a paralelização por múltiplas GPUs a qual também é levada em consideração no algoritmo. Para uso de **Múltiplas GPUs** cada espaço de trabalho é processado em uma GPU específica e ao final o resultado do processamento é consolidado, para isso se faz necessário a execução do kernel por diferentes threads do computador hospedeiro.

Cada thread é responsável por definir a GPU a ser utilizada, alocar e transferir os dados de entrada, e também realizar a chamada de execução do kernel. A implementação de múltiplas threads foi feita com a API de programação paralela OpenMP [3](Khronos Group [2013]), a qual é utilizada na criação e manipulação das diferentes threads de processamento. É criada uma thread para cada GPU disponível. O número de GPUs é informado como parâmetro de entrada da execução. O Algoritmo 6 mostra os passos executados no computador hospedeiro e o Algoritmo 7 mostra o kernel, executado em GPU. Além disso, a Figura 3.9 ilustra o processo de execução do kernel para múltiplas GPUs. As Tabelas 3.6 e 3.7 mostram uma breve documentação das funções utilizadas internamente nos algoritmos. Estas funções são utilizadas apenas como abstrações das operações realizadas, a implementação não utiliza necessariamente estas funções.

[3]http://www.openmp.org/

Algoritmo 6: Identificar Atratores em Paralelo (Hospedeiro)

Entrada: *nGPUs* (número de gpus a serem utilizadas), A (matriz de regulação), n (tamanho da matriz de regulação), *nGPUThreads* (número de threads da GPU a serem utilizadas)

Saída: *attrBySize* (tabela de espalhamento hash (chave/valor) onde para cada elemento, a *chave* é o identificador do atrator e o *valor* o tamanho da bacia de atração do mesmo).

```
1  // Executa em paralelo com nGPUs threads
2  ompThreadStart(nGPUs) ;
3  // Define a GPU a ser utilizada por esta thread
4  deviceId ← ompGetThreadId() ;
5  setDeviceId(deviceId) ;
6  // Obtém próximo espaço de trabalho disponível
7  offset ← 0 ;
8  workspaceSize ← getWorkSize(n, gpuMaxMemory()) ;
9  S ← getWorkspace(deviceId, n, workspaceSize, offset) ;
10 alocarEmGPU(Sd, workspaceSize) ; // Alocação de memória em GPU para Sd
11 // Tabela de espalhamento hash para thread corrente
12 // A chave é o identificador do atrator e o valor o tamanho da bacia de
      atração do mesmo
13 attrBySize ← initHash() ;
14 while S! = NULL do
15         copiarParaGPU(S, Sd) ;
16         // Executar em paralelo na GPU usando (nGPUThreads) threads
17         Cd ← Kernel Identificar Atratores em Paralelo
           (A, n, Sd, workspaceSize/nGPUThreads, offset) ;
18         copiarParaHost(Cd, C) ;
19         // Neste ponto, em C temos os atratores de cada estado de S
20         // Assincronamente, incrementa em uma unidade o valor de
              attrBySize para o respectivo atrator em C
21         consolidarAssync(attrBySize, S, C) ;
22         // Obtém próximo espaço de trabalho disponível para a
              thread
23         offset ← offset + workspaceSize
           workspaceSize ← getWorkSize(n, gpuMaxMemory()) ;
24         S ← getWorkspace(deviceId, n, workspaceSize, offset)
25 end
26 desalocarDaGPU(Sd) ;
27 // Consolidação final do resultado entre as diferentes threads do host
      (garante atomicidade)
28 // Considerar attrBySizeResult visível para todas as threads do host
29 consolidar(attrBySizeResult, attrBySize) ;
30 // Marca o fim da execução de múltiplas threads
31 ompThreadJoin() ;
32 return attrBySizeResult ;
```

Função	Descrição
alocarEmGPU	Aloca memória na GPU
desalocarEmGPU	Libera memória alocada na GPU
ompThreadStart	Inicializa as múltiplas threads do computador hospedeiro
ompGetThreadId	Obtém o identificador da thread corrente (de $0, \ldots, nGPUs$)
setDeviceId	Define a GPU a ser utilizada na thread corrente
gpuMaxMemory	Obtém a quantidade máxima de memória da GPU
getWorkSize	Obtém o tamanho do espaço de trabalho para a thread corrente
getWorkspace	Obtém o espaço de trabalho para a thread corrente
consolidarAssync	Efetua consolidação assíncrona dos resultados em uma tabela hash
consolidar	Efetua consolidação dos resultados em uma tabela hash

Tabela 3.6: *Documentação das funções usadas no Algoritmo 6. Estas funções abstraem as operações descritas e não serão detalhadas.*

Algoritmo 7: Kernel Identificar Atratores em Paralelo (GPU)

Entrada: A (matriz de regulação gênica), n (tamanho da matriz de regulação - $A_{n \times n}$), S (vetor de estados do grafo na memória da GPU), k (tamanho do vetor de estados), $offset$ (deslocamento para identificação do estado atual de processamento)

Saída: C identificação dos atratores para cada estado de S.

```
1  tid ← getThreadID() ; // Identificador da thread obtido do ambiente CUDA.
2  state ← nextInitialState(tid, S, k, 0, offset) ; // Estado a ser processado na thread,
   será -1 caso não existam mais estados.
3  stateIdx ← idxFromState(tid, state, k) ; // Índice do estado atual no vetor saída.
4  stack ← initStack(MAX-STACK-SIZE) ;
5  while state ≠ −1 do
6  |   clean(stack) ;
7  |   while notfoundAttractor(stack, state) do
8  |   |   push(stack, state) ;
9  |   |   state ← nextState(A, n, state);
10 |   end
11 |   C[stateIdx] ← state ;
12 |   state ← nextInitialState(tid, S, k, state, offset) ;
13 |   stateIdx ← idxFromState(tid, state, k) ;
14 end
15 return C ;
```

Função	Descrição
getThreadID	Obtém o identificador da thread corrente na GPU
initStack	Inicializa uma estrutura de dados de pilha
idxFromState	Índice do estado informado no vetor saída
nextState	Obtém o próximo estado da rede após a aplicação da matriz de regulação
nextInitialState	Obtém o próximo estado inicial a ser processado
foundAttractor	Verifica se existe um ciclo no caminho corrente armazenado pilha informada

Tabela 3.7: *Documentação das funções usadas no Algoritmo 7. Estas funções abstraem as operações descritas e não serão detalhadas.*

O Algoritmo 6 recebe como parâmetros de entrada:

- *nGPUs*: número de GPUs a serem utilizadas;
- *A*: matriz de regulação gênica e suas dimensões $n \times n$;
- *nGPUThreads*: número de threads da GPU a serem utilizadas.

Como saída fornece uma tabela de espalhamento *hash* (chave/valor), onde para cada elemento temos:

- *chave:* identificador do atrator no formato decimal;
- *valor:* tamanho da bacia de atração do atrator armazenado na chave.

O número de GPUs (*nGPUs*) é necessário para definir a quantidade de GPUs a serem utilizadas para execução do algoritmo e o número de threads (*nGPUThreads*) é utilizado na chamada para execução do kernel que realiza a identificação dos atratores dado um espaço de trabalho (Algoritmo 7). Na prática o número de threads é definido pelo tamanho de *grid* e bloco, porém abstraímos estes valores no algoritmo para o número total de threads. Estes parâmetros também são utilizados na determinação do tamanho do espaço de trabalho de cada GPU, sendo o espaço de estados dividido entre as mesmas, gerando um espaço de trabalho para cada. Caso o tamanho do espaço de trabalho ultrapasse a memória máxima da GPU o mesmo é quebrado em diversas partes, para isso, a função *getWorkSize* se vale da quantidade máxima de memória disponível da GPU e do número de estados a ser processado, para calcular o tamanho máximo de espaço de trabalho possível.

Na linha 2 a função *ompThreadStart* indica a criação de *nGPUs* threads. A partir deste ponto, temos a execução de múltiplas threads do computador hospedeiro, uma para cada GPU, até a linha 31, onde a função *ompThreadJoin* efetua a finalização destas threads. Para cada thread, é indicada a GPU a ser utilizada pela função *setDeviceId*. A variável $offset$ armazena o deslocamento a ser considerado para geração dos estados no espaço de trabalho em questão. Cada parte do espaço de trabalho a ser processada é obtida pela função *getWorkspace* e é armazenado na variável *S*. O laço da linha 14 efetua o processamento de todas as partes, sendo os novos espaços de trabalho a serem processados atualizados na linha 24. A função *getWorkspace* devolve *NULL* quando não existirem mais partes do espaço de trabalho a serem processadas. Para o processamento em GPU, *S* é copiado para memória da GPU, sendo armazenado na variável *Sd*. Esta cópia é feita pela função *copiarParaGPU*, na linha 15 e após sua realização, o kernel que implementa o Algoritmo 7 é executado na linha 17 utilizando *nGPUThreads*.

Um detalhe importante é que o espaço de trabalho, para ser processado na GPU, é dividido entre as threads (*workspaceSize*/*nGPUThreads*). Após a execução do kernel, o resultado é armazenado na memória da GPU na variável *Cd*, a qual contém a identificação dos atratores de cada um dos estados da área de trabalho (ou sua parte) processada pela GPU. O resultado é copiado para memória RAM, na linha 18, sendo armazenado na variável *C*. Neste ponto é importante notar que *C* é visível apenas para a thread (do computador hospedeiro) corrente. Na implementação, o processamento do resultado foi é feito de forma assíncrona (isto é por uma thread separada). Este fato é abstraído no algoritmo, com a chamada da função *consolidarAssync* na linha 21, a qual realiza a consolidação dos atratores de *C* na tabela de espalhamento hash *attBySize*. A consolidação é feita de forma simples: para cada atrator de *C* adiciona-se o mesmo como chave na tabela hash, se o mesmo não existe, inicializa o valor deste na tabela com 1, caso já exista, incrementa-se em uma unidade o valor da tabela para a chave em questão. A consolidação assíncrona permite novas execuções do kernel ao mesmo tempo que o resultado é consolidado.

Quando não há mais nenhum estado do espaço de trabalho a ser processado, o laço termina e na linha 26 ocorre a liberação de recursos alocados inicialmente na memória da GPU. Por fim, na

linha 29, com a execução da função *consolidar* os resultados obtidos entre diversas GPUs (múltiplas threads no computador hospedeiro) são consolidados na variável *attrBySizeResul*. Esta variável é compartilhada entre as múltiplas threads. A função consolidar garante a consistência desta variável durante a sua execução mesmo se executada por múltiplas threads (simplesmente serializa as chamadas entre múltiplas threads). A consolidação é feita de forma similar à anterior, porém ao invés de incrementar em uma unidade para cada atrator localizado, os valores contidos nas tabelas informadas são somados em *attrBySizeResul*.

Na Figura 3.9 temos duas GPUs identificadas por *deviceId* 0 e *deviceId* 1. Estas GPUs executam o kernel do Algoritmo 7 utilizando $p = nGPUThreads$, identificadas de $0, \ldots, (p-1)$. Cada thread da GPU processa uma pequena parte do espaço de trabalho em questão, de tamanho definido pelo parâmetro de entrada k, onde neste caso, $k = workspaceSize/nGPUThreads$, ou seja o espaço de trabalho é dividido para ser processado entre as inúmeras threads da GPU. Ainda nesta mesma figura, vemos que a *thread [0]* processa os estados identificados na forma decimal de 0 até $(k-1)$, já a *thread [1]* de k até $(2k-1)$, e assim, sucessivamente, até todos os estados do espaço de trabalho serem processados.

No Algoritmo 7 a identificação da thread é obtida utilizando a função *getThreadId* (linha 2). Este valor é utilizado para obtenção do estado inicial a ser processado, obtido pela função *nextInitialState*, na thread de identificador *tid* - variável que armazena o identificador da thread. Após isso, na função *idxFromState* é obtido o índice (*stateIdx*) do estado a ser processado no vetor saída. Este índice é o local onde será armazenado o atrator deste estado. Para identificação dos atratores é utilizada uma estrutura de dados de pilha. No laço da linha 7 vemos que cada thread efetua um caminhamento, até o atrator ser identificado. A pilha é utilizada para verificar se existe um ciclo no caminho efetuado, essa verificação é feita pela função *foundAttractor*, que recebe a pilha e o estado corrente do caminhamento. O próximo estado é definido na linha 9, pela função *nextState*, a qual efetua a multiplicação de matriz, seguida de limiarização dada pela Equação 2.5. Ao encontrar o atrator, o laço da linha 7 finaliza. A identificação do atrator é armazenada no índice *stateIdx*, obtido anteriormente, que representa o estado inicial do caminhamento. O processo é repetido até não existirem mais estados a serem processados.

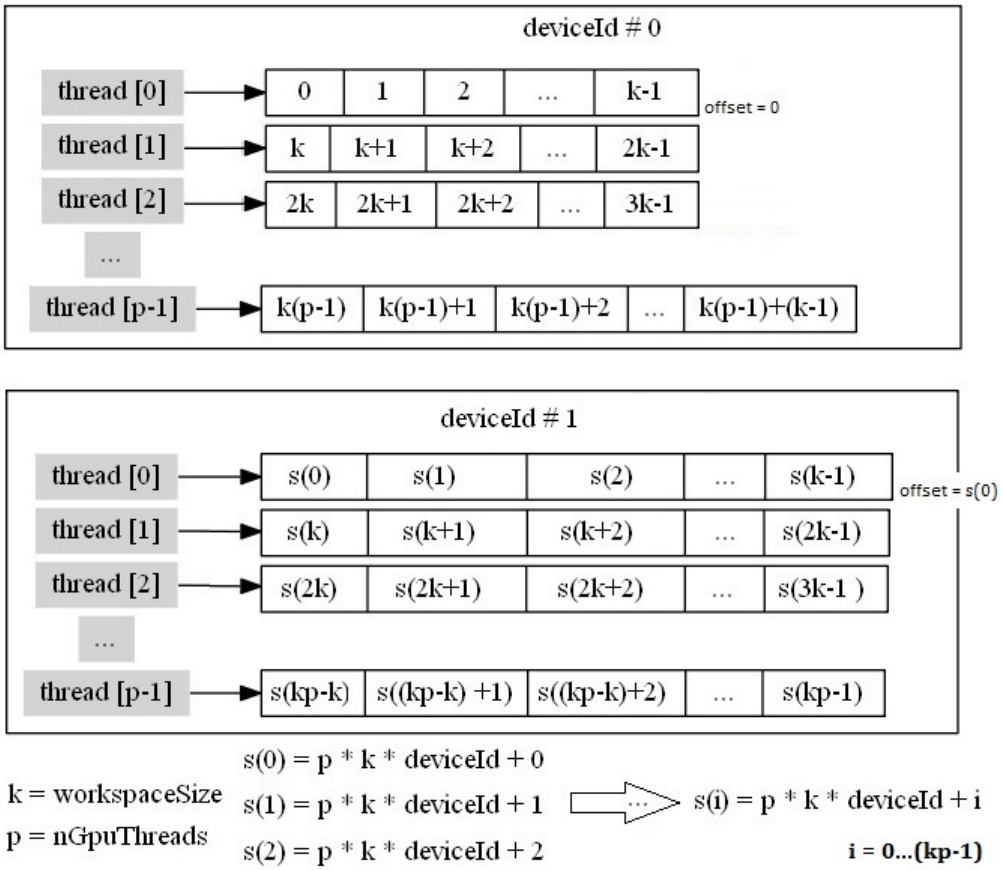


Figura 3.9: *Esquema de execução das threads e áreas de trabalho do Algoritmo 6. Cada uma das p GPUs (executando em uma thread específica do host - deviceId* 0 *e* 1*) é responsável por uma área de trabalho. Para execução do kernel a área de trabalho é dividida para ser processada pelas inúmeras threads da GPU. Cada thread processa um trecho da área de trabalho (de tamanho igual a k - tamanho da área de trabalho), determinando o atrator para cada estado nela contido. Ao finalizar o processamento de para um estado, avança para o próximo, até todos os estados serem processados.*

Capítulo 4

Resultados

Neste capítulo apresentamos alguns resultados obtidos com a implementação. Para isso, são exibidas informações sobre os experimentos realizados, com relação ao tempo de execução, para redes geradas de forma aleatória. Começamos mostrando na Seção 4.1 alguns resultados obtidos no trabalho de Andrade [2012], onde se obteve ganhos de até 20x no tempo de execução para processar redes de tamanho 20. Após isso, na Seção 4.2, são exibidos gráficos relativos a comparação dos tempos de execução e desempenho entre os diferentes métodos estudados.

4.1 Utilização da implementação em outros trabalhos

Durante o desenvolvimento deste trabalho foi disponibilizada uma versão inicial da implementação em GPU do Algoritmo 2 ("Identificação de Atratores utilizando Componentes Conexas em Grafos") para ser utilizada no trabalho de Andrade [2012], trazendo ganhos de desempenho com redução do tempo gasto nas simulações das redes geradas pelo algoritmo do referido trabalho.

Os experimentos foram feitos em uma GPU NVIDIA GeForce GT330M com 512 MB de VRAM, em um MacBook Pro6.2, com processador Intel Core i7, com clock de 2.66GHz com dois núcleos e 8GB de memória RAM. Esta GPU possui 48 núcleos CUDA, podendo rodar até 512 threads simultâneas, com um clock de 1.1GHz. A Tabela 4.1 mostra a média do tempo gasto para algumas simulações neste ambiente (Andrade [2012], p32).

Tamanho da rede	Tempo Médio Impl. Geral Serial	Tempo Médio Impl. em CUDA
11	3s	40ms
20	60s	3s

Tabela 4.1: *Resultados do uso da implementação em outros trabalhos. Testes realizados no trabalho de Andrade [2012], para 50 redes de cada tamanho, geradas por seu algoritmo de inferência.*

4.2 Análise de Desempenho

Os experimentos deste trabalho foram realizados utilizando um ambiente *Linux Ubuntu 11.04 64bits*, com processador Intel® Core™ i7 com 2.67GHz sendo 8 núcleos físicos (Hyper-threading - 2 threads por núcleo) e duas placas NVIDIA, com as seguintes configurações:

- GeForce GTX 470, CUDA capability 2.0, com 448 cores e 1280MB VRAM;
- GeForce GTX 275, CUDA capability 1.3, com 240 cores e 896MB VRAM.

Foi realizado um comparativo entre as implementações:

- Implementação geral sequencial em CPU;
- Implementação utilizando resolvedor SAT [1];
- Implementação por componentes conexas de Grafo em GPU;
- Implementação por trajetórias em paralelo em GPU.

Por questões de apresentação nos gráficos, para nomear os diferentes métodos, adotamos a convenção da Tabela 4.2.

Método	Nomenclatura
Implementação geral sequencial em CPU	Serial
Implementação utilizando resolvedor SAT	SAT
Implementação por componentes conexas de Grafo em GPU	Graph GPU
Implementação por trajetórias em paralelo em GPU	Parallel GPU

Tabela 4.2: *Nomenclatura para os diferentes métodos comparados.*

4.2.1 Redes Utilizadas

Neste trabalho não estamos diretamente interessados em critérios para geração de redes e sim na obtenção dos atratores e tamanho das bacias de atração. Porém, para uso nos experimentos é interessante que as redes possuam características biológicas. As redes utilizadas foram geradas de forma aleatória. Como parâmetro da geração das redes temos o *tamanho da rede* (definido por n) e o *grau máximo de entrada dos genes* (ou conectividade máxima k). O tamanho da rede define as dimensões $n \times n$ da matriz de regulação, na qual cada linha contém no máximo k elementos com valores diferentes de zero. O grau máximo de entrada (k) define o número máximo de preditores que um gene. Nos experimentos utilizamos valores de $k = \{2, 6, 8\}$. Especificamente utilizamos $k = 2$, baseando-se no trabalho de Kauffman [1969], que diz que as redes gênicas estão na "fronteira entre a ordem e o caos"(redes com número de preditores entre 2 e 3). Os outros valores $k = 6$ e $k = 8$ foram utilizados para verificar escalabilidade e comparação entre os métodos.

O procedimento para geração das redes é o seguinte:

1. Inicializar a matriz A com todas as posições iguais a 0;
2. Para cada linha j da matriz, $j = 0, \ldots, (n-1)$, sorteia-se, sem repetição, um conjunto $P_j = (p_0, p_1, \ldots, p_{(k-1)})$, $p_i = 0, \ldots, (n-1)$, contendo k elementos escolhidos entre $\{0, 1, \ldots, (n-1)\}$;
3. Para cada elemento p_i de P, efetua-se um novo sorteio de um valor v entre $\{-1, 0, 1\}$ (se $p_i \neq j$) ou $\{-1, 0\}$ (se $p_i = j$) e define $A[p_i][j] = v$.

Por exemplo, seja $n = 10$, $k = 6$. Queremos obter a matriz de regulação T. Inicialmente todas as posições de T são iguais a zero. Para linha 0 ($j = 0$) efetuamos um sorteio de 6 ($k = 6$) valores do conjunto $\{0, 1, \ldots, 9\}$, obtendo como resultado o conjunto $P_0 = \{1, 4, 7, 9, 3, 6\}$. Para cada valor i de P_0, se $i \neq j$, efetuamos um novo sorteio entre os valores $\{-1, 0, 1\}$, se $i = j$ o sorteio é realizado entre os valores $\{-1, 0\}$, obtendo-se v. Seja $i = 1$, pelo sorteio obtemos $v = -1$, portanto $T[i][j] = T[1][0] = -1$. O mesmo é feito para todos os outros 5 elementos de P_0 até completar a primeira linha da matriz e assim sucessivamente para configurar todas as outras 9 linhas restantes.

Na geração das matrizes, os sorteios são realizados segundo uma distribuição uniforme de probabilidade, obtendo como resultado, matrizes aleatórias com conectividade máxima igual à informada.

[1] http://web.it.kth.se/ dubrova/bns.html

4.2.2 Experimentos

Os experimentos levam em consideração o tempo de execução para 10 grupos de redes. Os grupos são identificados pelo número de genes, variando de 20 a 30. Cada grupo possui 60 redes, 20 para cada configuração de grau de entrada máximo $k = \{2, 6, 8\}$. Para cada método (*Serial*,*SAT*, *Graph GPU* e *Parallel GPU*), foram realizadas medições dos tempos de execução para as redes geradas de forma aleatória, segundo a configuração de cada grupo. As medições de cada método são exibidas em gráficos individuais (um por grau de entrada), informando o desvio padrão da amostra. Em seguida, as medições são apresentadas em conjunto, para análise do efeito do aumento do grau de entrada no tempo de execução. Após apresentação detalhada de cada método, é feita uma comparação entre os mesmos para análise do desempenho. O método *Parallel GPU* será comparado detalhadamente com relação a cada um dos outros métodos.

Método *Serial*

A Figura 4.1 mostra o tempo médio de execução (em segundos) para o método *Serial* para redes com grau máximo de entrada $k = 2$, inclusive o desvio padrão da amostra. A mesma informação, para redes com $k = 6$ e $k = 8$, pode ser vista nas figuras 4.2 e 4.3, respectivamente. Podemos ver que o desvio padrão das amostras é baixo, mostrando que o tempo de execução não tem variações abruptas e desta forma é bem controlado.

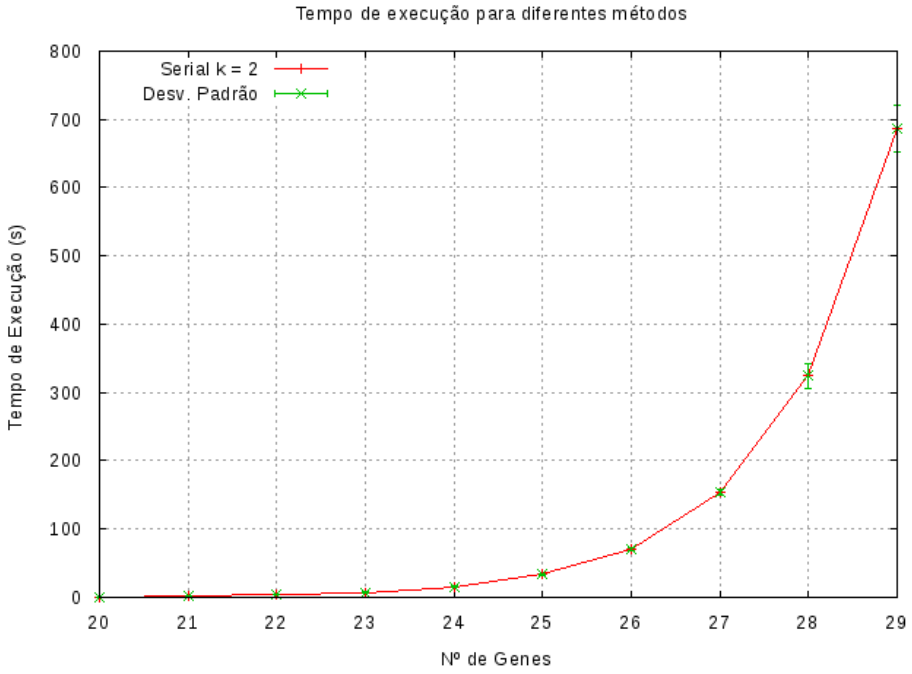


Figura 4.1: *Tempo de execução do método* Serial *(grau máximo de entrada $k = 2$) e desvio padrão.*

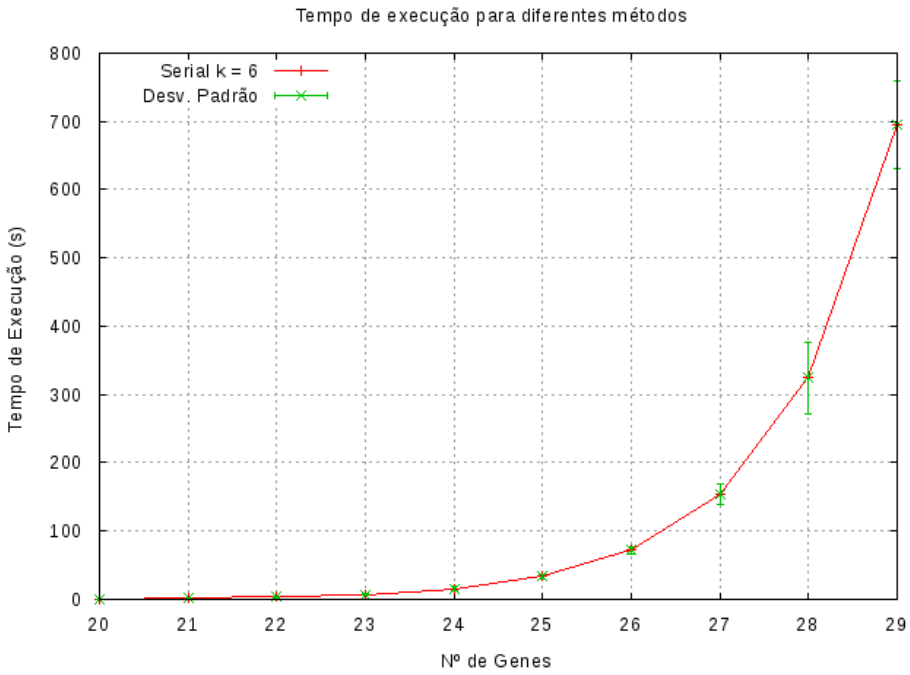


Figura 4.2: *Tempo de execução do método* Serial *(grau máximo de entrada* $k = 6$*) e desvio padrão.*

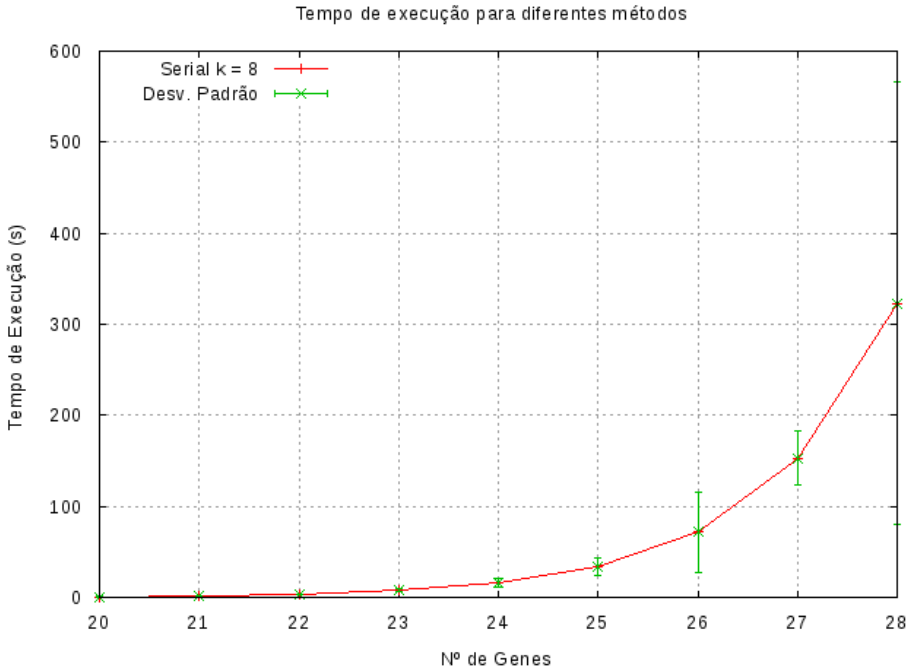


Figura 4.3: *Tempo de execução do método* Serial *(grau máximo de entrada* $k = 8$*) e desvio padrão.*

A Figura 4.4 mostra em conjunto as curvas, do método para todas as configurações utilizadas ($k = \{2, 6, 8\}$). Vemos que o método *Serial* não varia com relação a este parâmetro, mantendo-se praticamente igual, mesmo para redes com diferentes valores de k.

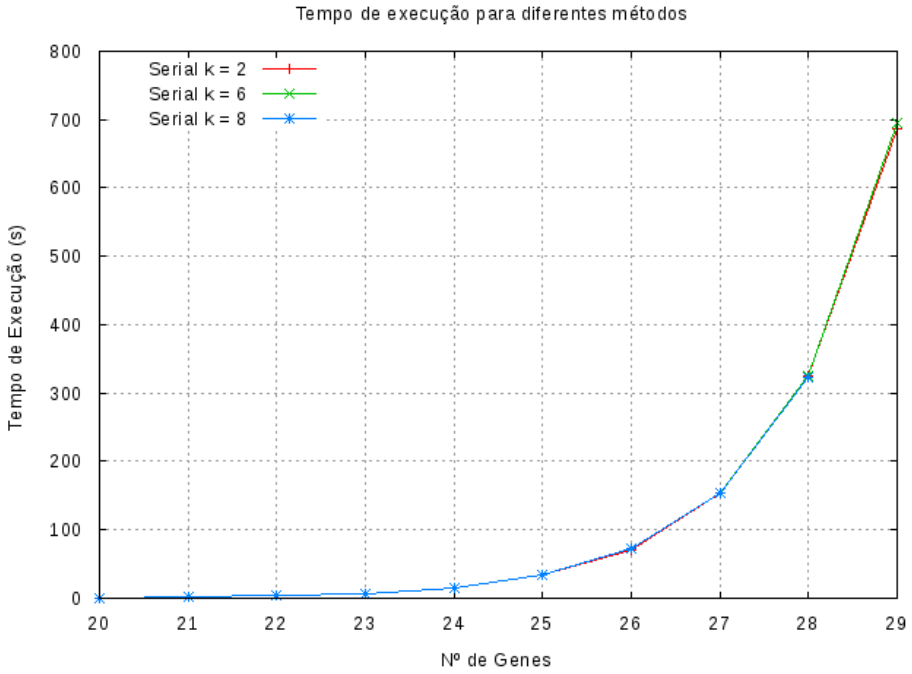


Figura 4.4: *Tempo de execução do método* Serial. *São exibidos os tempos de execução para redes com grau máximo de entrada* $k = \{2, 6, 8\}$. *Mesmo com alteração do grau e entrada, as curvas para* $k = 2$, $k = 6$ *e* $k = 8$, *se mantém praticamente idênticas.*

Método *SAT*

A Figura 4.5 mostra o tempo médio de execução (em segundos) para o método *SAT* para redes com grau máximo de entrada $k = 2$, inclusive o desvio padrão da amostra. É importante lembrar que este método obtém somente os atratores e não efetua contagem do tamanho das bacias de atração. A mesma informação, sobre o tempo médio de execução, para redes com $k = 6$ e $k = 8$, pode ser vista nas figuras 4.6 e 4.7, respectivamente. O desvio padrão das amostras é relativamente alto, mostrando que o tempo de execução sofre variações abruptas e desta forma não é tão bem controlado (em relação aos outros métodos, no que diz respeito ao desvio padrão). Mesmo assim, é possível observar, principalmente para $k = 2$, que o tempo de execução mantém uma taxa de crescimento baixa com o aumento no número de genes.

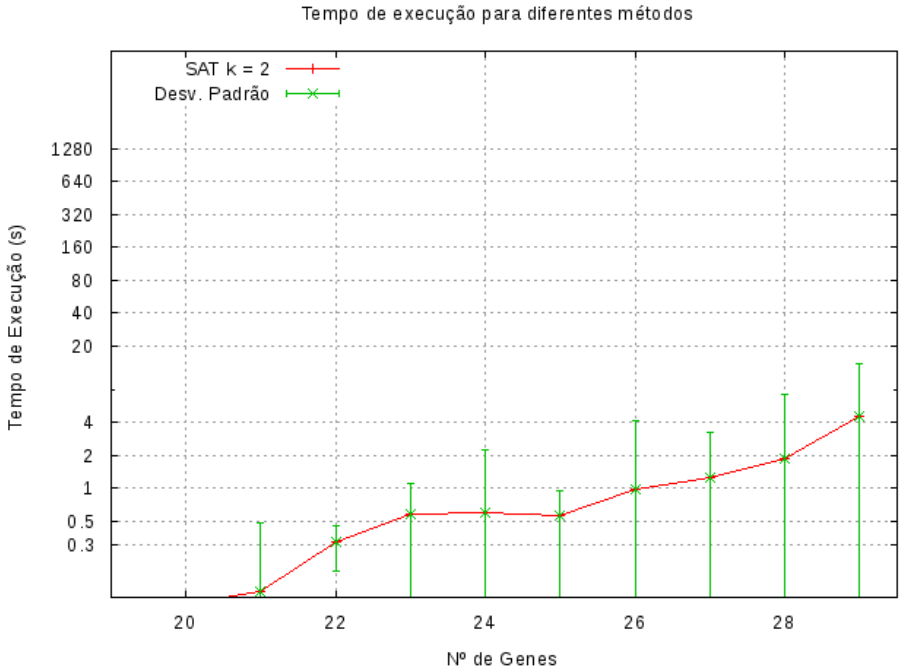


Figura 4.5: *Tempo de execução do método* SAT *(grau máximo de entrada* $k = 2$*) e desvio padrão.*

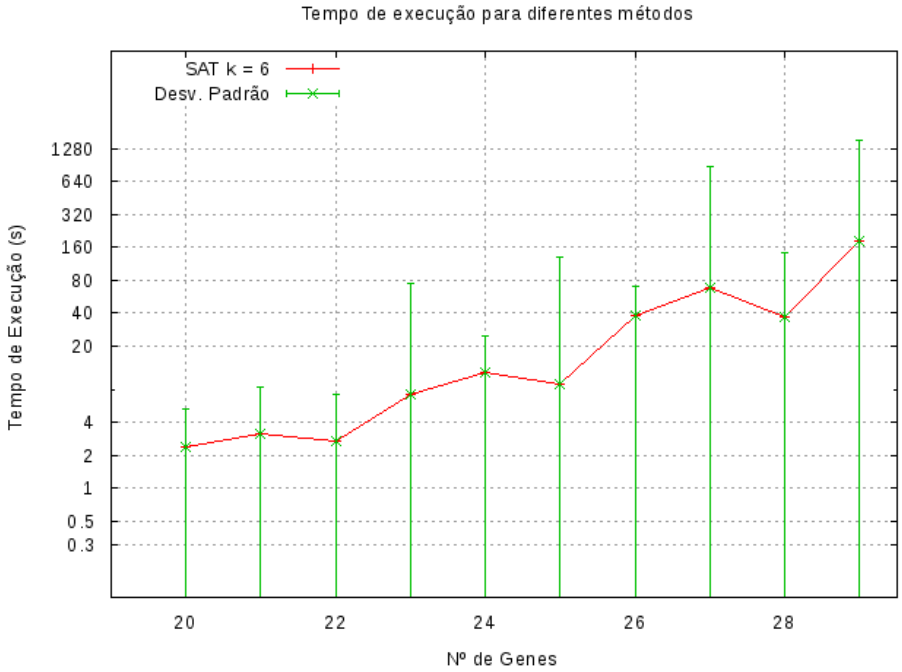


Figura 4.6: *Tempo de execução do método* SAT *(grau máximo de entrada* $k = 6$*) e desvio padrão.*

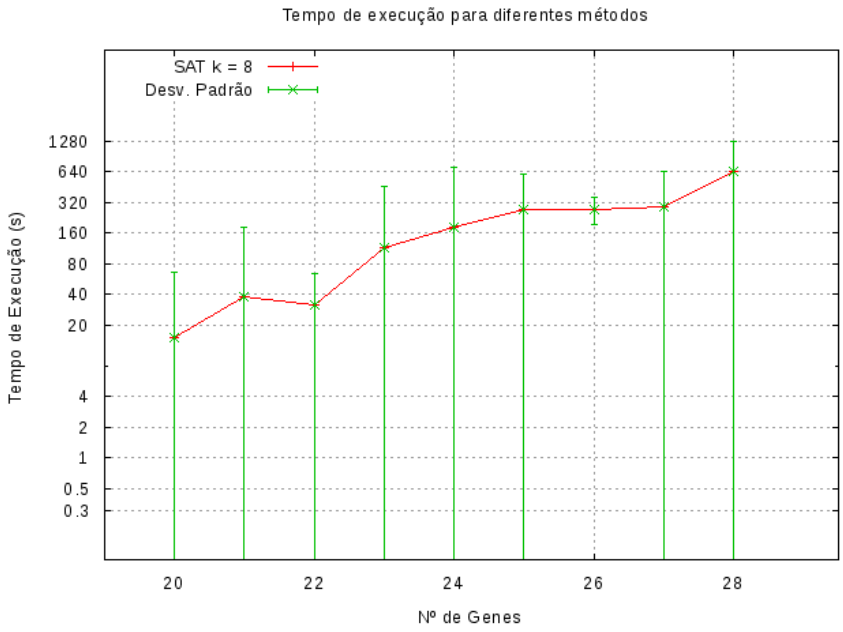


Figura 4.7: *Tempo de execução do método* SAT *(grau máximo de entrada* $k = 8$*) e desvio padrão.*

A Figura 4.8 mostra em conjunto as curvas para todas as configurações utilizadas ($k = \{2, 6, 8\}$), com escala logarítmica para melhor visualização dos dados. A comparação para diferentes valores de k mostra que o método é sensível a este parâmetro, o que causa um aumento considerável na média do tempo de execução (crescimento em torno de 10×). É interessante notar que para $k = 2$ o tempo de processamento é consideravelmente reduzido, chegando, em média, a 4 segundos para redes com 29 genes (o método *Serial* leva em torno de 600 segundos). Mesmo sem obter o *tamanho das bacias de atração*, o método *SAT* mostra-se interessante para obtenção dos atratores para redes com baixo grau médio de entrada.

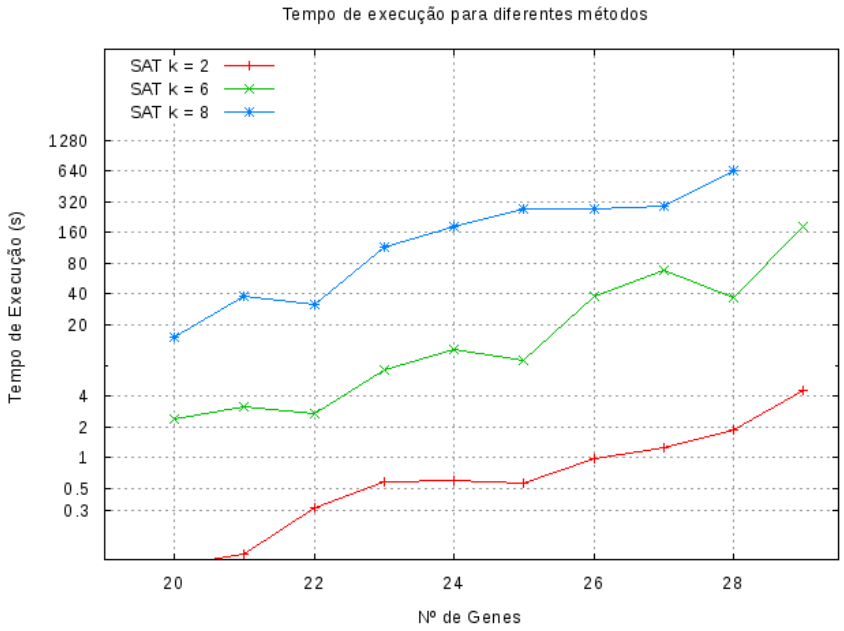


Figura 4.8: *Tempo de execução do método* SAT. *São exibidos os tempos de execução para redes com grau máximo de entrada* $k = \{2, 6, 8\}$. *O tempo para redes com* $k = 2$ *é consideravelmente reduzido, mostrando a eficiência deste método para redes com esta característica (grau de entrada baixo).*

Método *Graph GPU*

O método *Graph GPU* não suporta a utilização de múltiplas GPUs. Nos experimentos foi utilizada a GPU *GeForce GTX* 470. A Figura 4.9 mostra o tempo médio de execução (em segundos) para o método para redes com grau máximo de entrada $k = 2$, inclusive o desvio padrão da amostra. A mesma informação, para redes com $k = 6$ e $k = 8$, pode ser vista nas figuras 4.10 e 4.11, respectivamente. O desvio padrão das amostras é baixo, mostrando que o tempo de execução não sofre variações abruptas e desta forma é bem controlado.

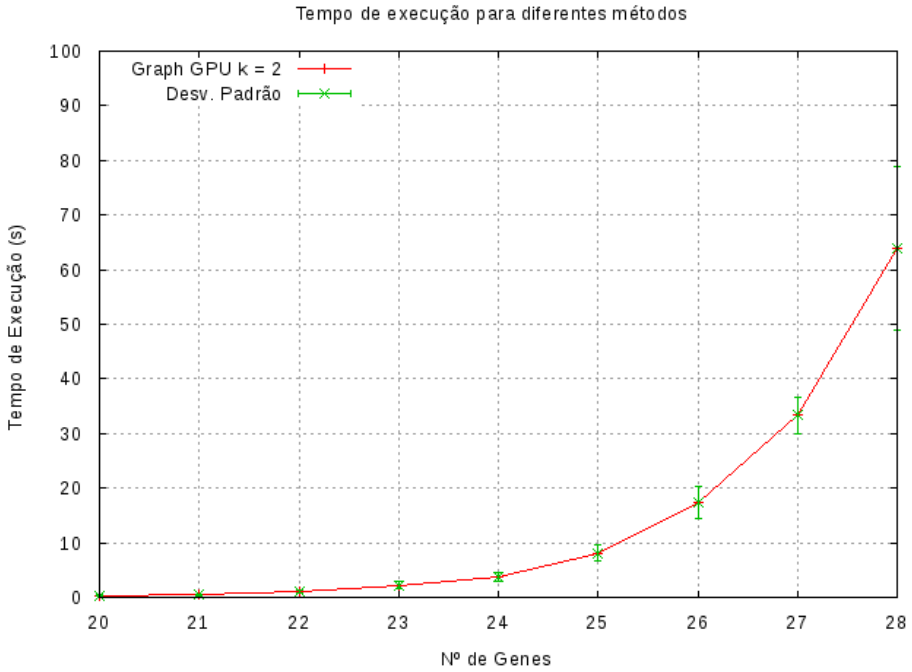


Figura 4.9: *Tempo de execução do método* Graph GPU *(grau máximo de entrada $k = 2$) e desvio padrão.*

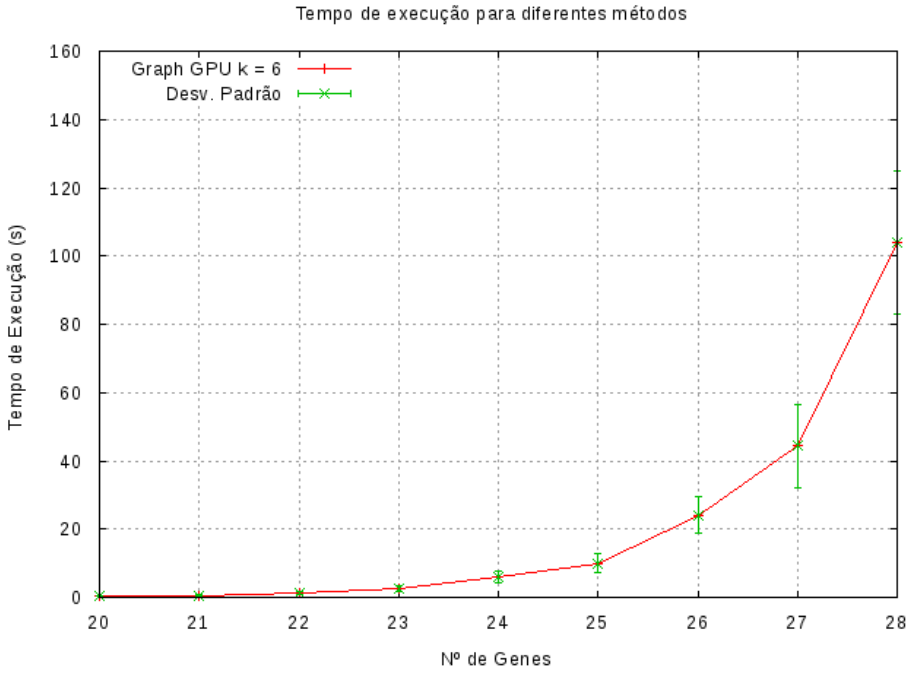


Figura 4.10: *Tempo de execução do método* Graph GPU *(grau máximo de entrada $k = 6$) e desvio padrão.*

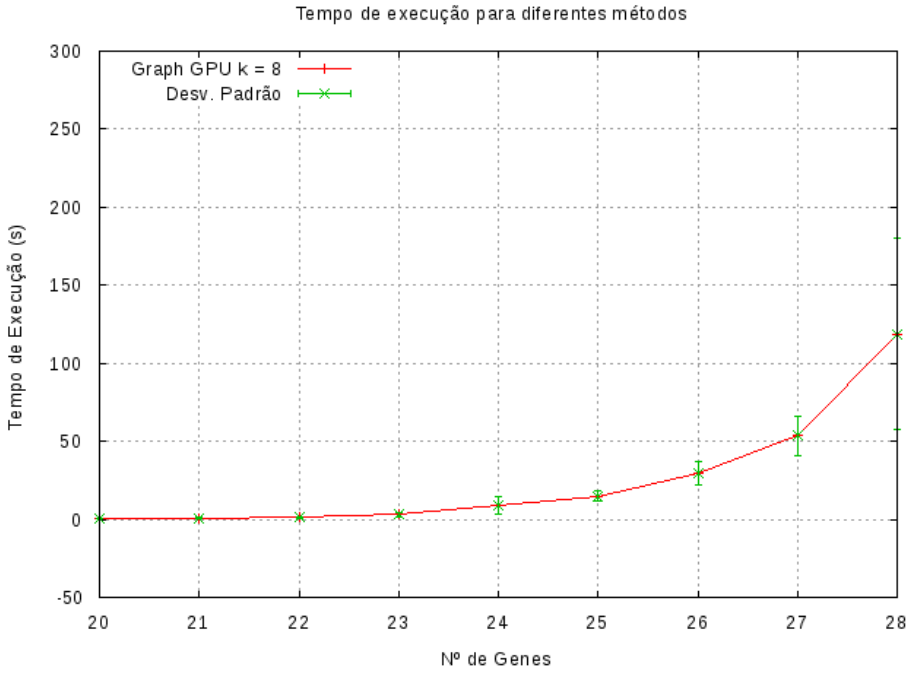


Figura 4.11: *Tempo de execução do método* Graph GPU *(grau máximo de entrada* $k = 8$*) e desvio padrão.*

A Figura 4.12 mostra em conjunto todas as configurações utilizadas ($k = \{2, 6, 8\}$). Nesta figura, vemos uma pequena variação no tempo de execução com o aumento de k (porém em proporção relativamente menor que no método *SAT*).

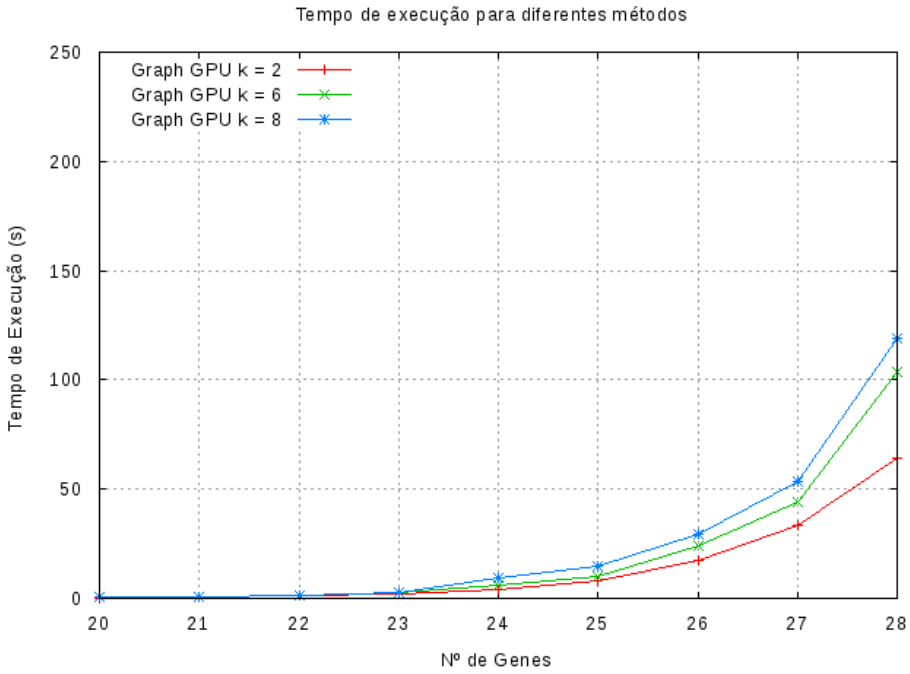


Figura 4.12: *Tempo de execução do método* Graph GPU*. São exibidos os tempos de execução para redes com grau máximo de entrada* $k = \{2, 6, 8\}$*.*

Método *Parallel GPU*

Para o método *Parallel GPU*, os experimentos foram realizados utilizando apenas uma GPU, a *GeForce GTX 470*. A Figura 4.13 mostra o tempo médio de execução (em segundos) para redes com grau máximo de entrada $k = 2$, inclusive o desvio padrão da amostra. A mesma informação, para redes com $k = 6$ e $k = 8$, pode ser vista nas figuras 4.14 e 4.15, respectivamente. O desvio padrão das amostras é baixo, mostrando que o tempo de execução não sofre variações abruptas e desta forma é bem controlado.

Figura 4.13: *Tempo de execução do método* Parallel GPU *(grau máximo de entrada $k = 2$) e desvio padrão.*

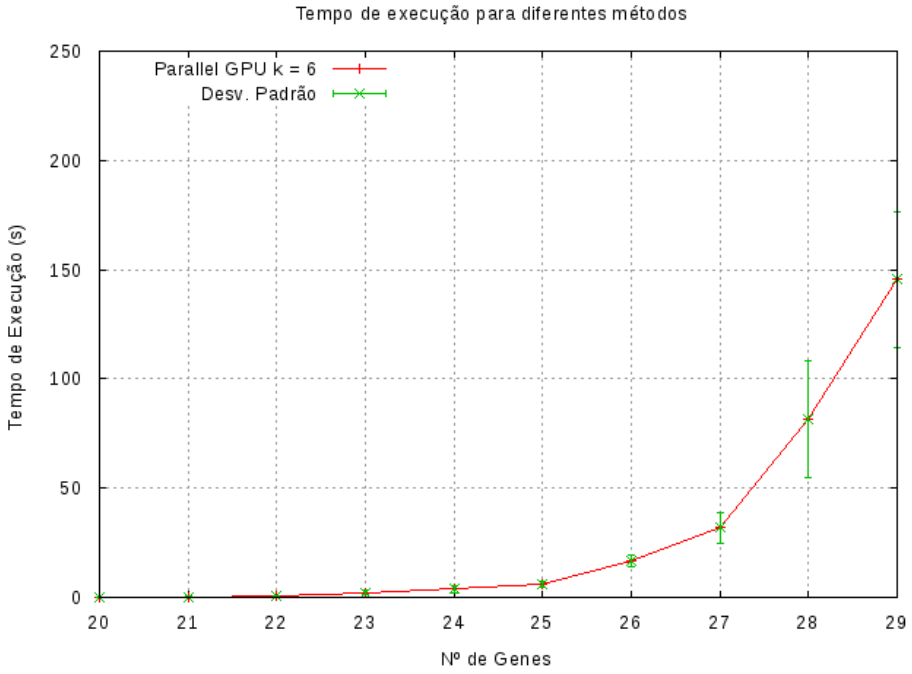


Figura 4.14: *Tempo de execução do método* Parallel GPU *(grau máximo de entrada $k = 6$) e desvio padrão.*

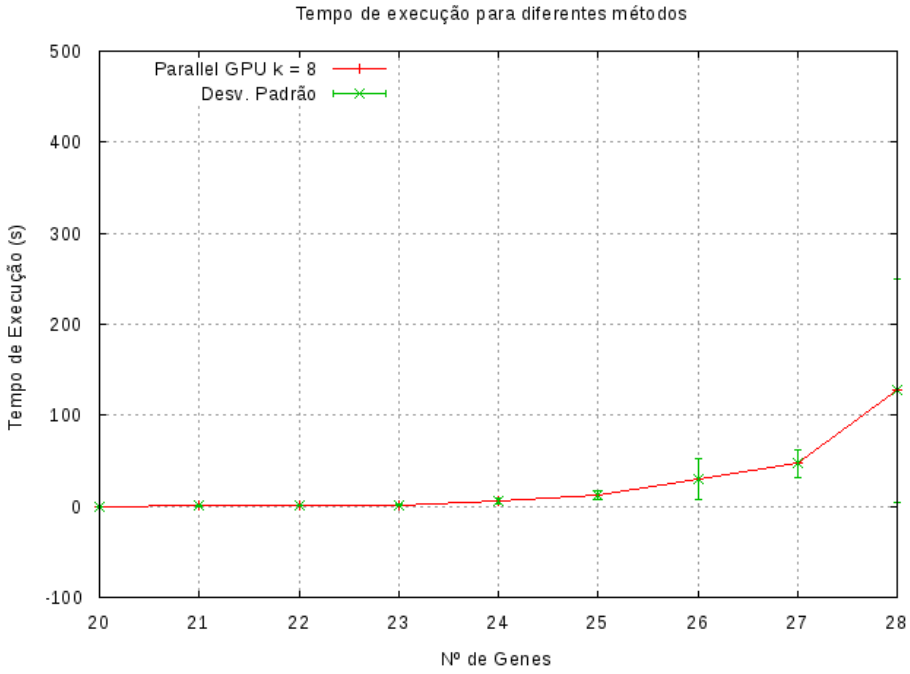


Figura 4.15: *Tempo de execução do método* Parallel GPU *(grau máximo de entrada* $k = 8$*) e desvio padrão.*

A Figura 4.16 mostra em conjunto todas as configurações utilizadas ($k = \{2, 6, 8\}$), vemos também um aumento no tempo de execução com o aumento de k (porém em proporção menor que no método *SAT* e um pouco maior que no método *Graph*).

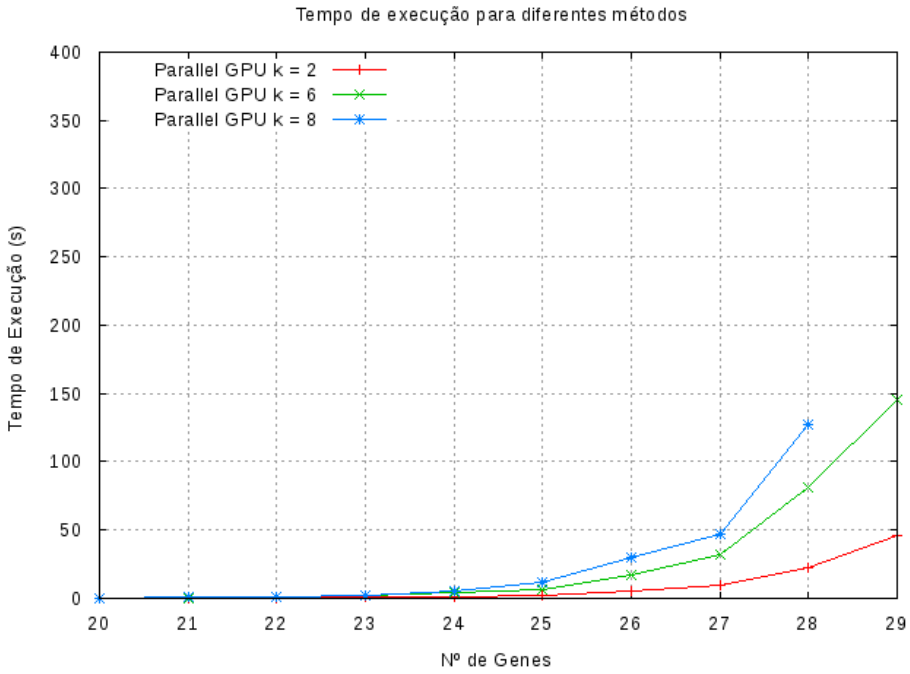


Figura 4.16: *Tempo de execução do método* Parallel GPU. *São exibidos os tempos de execução para redes com grau máximo de entrada* $k = \{2, 6, 8\}$.

Comparação dos Tempos de Execução

As figuras 4.8, 4.12 e 4.16, exibidas anteriormente, mostram o tempo médio de execução, para os métodos *Graph GPU*, *SAT* e *Parallel GPU*, respectivamente, realizando um comparativo com relação às diferentes configurações de grau máximo de entrada ($k = \{2, 6, 8\}$), utilizadas nos experimentos. É interessante notar que o método *Serial*, mesmo com a variação do grau máximo de entrada, manteve a mesma média do tempo de execução para diferentes tamanhos de rede. Outra visão do tempo médio de execução, agora realizando um comparativo entre os métodos, utilizando as mesmas configurações de grau de máximo de entrada, pode ser vista nas figuras 4.17, 4.18 e 4.19. Cada figura refere-se à uma configuração específica, onde todos os métodos são exibidos em conjunto. Utilizamos escala logarítmica para melhor visualização dos dados.

Na Figura 4.17 temos os dados para configuração $k = 2$. É possível verificar que o método *Serial* tem o tempo de execução mais elevado, enquanto o método *SAT*, em geral, para todos os tamanhos de rede utilizados, tem o menor tempo. Neste gráfico fica visível que o método *SAT*, para redes com grau de entrada baixo, por exemplo, o valor utilizado $k = 2$, mostra bons tempos de execução na tarefa de identificação dos atratores. Os métodos *Graph GPU* e *Parallel GPU* também mostram bons tempos de execução, sendo o tempo do método *Graph GPU* em média 10× menor que o tempo do *Serial* e o tempo do *Parallel GPU* até 20× menor que o tempo do *Serial*.

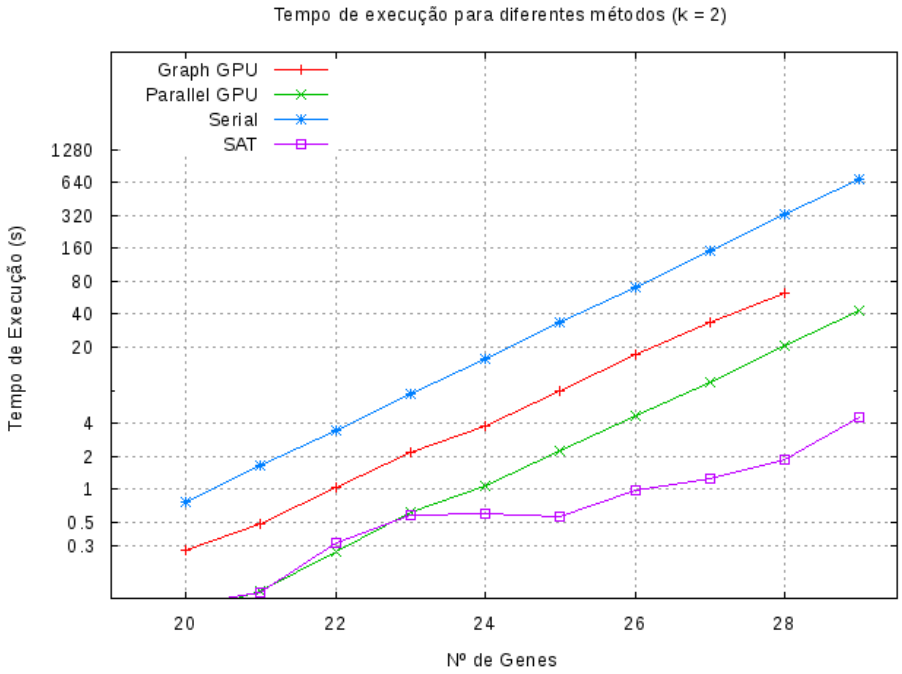


Figura 4.17: *Comparação entre os diferentes métodos ($k = 2$)*

Para as figuras 4.18 e 4.19 vemos um crescimento considerável no tempo de execução do método *SAT*, isso ocorre pois o problema de satisfatibilidade booleana fica cada vez mais complexo de ser resolvido, devido ao crescimento no número de variáveis (pelo aumento gradual de k). O tempo do método *Serial* se mantém praticamente idêntico para os diferentes valores de k. Já para os métodos *Graph GPU* e *Parallel GPU* ocorre uma leve movimentação das curvas (no eixo y), mostrando que também há dependência deste parâmetro, porém não tão elevada quanto no método *SAT*.

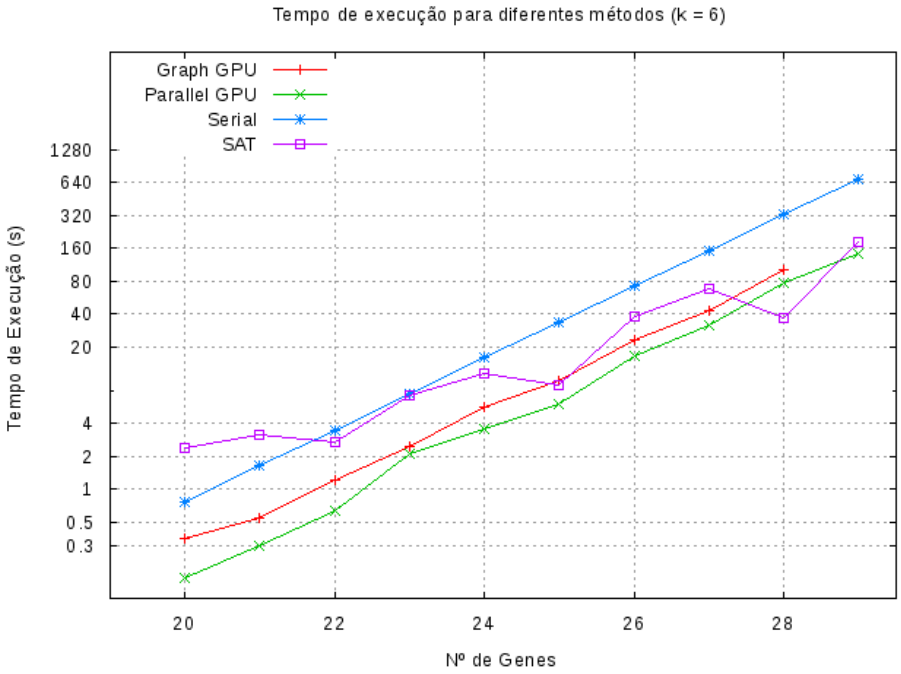


Figura 4.18: *Comparação entre os diferentes métodos ($k = 6$)*

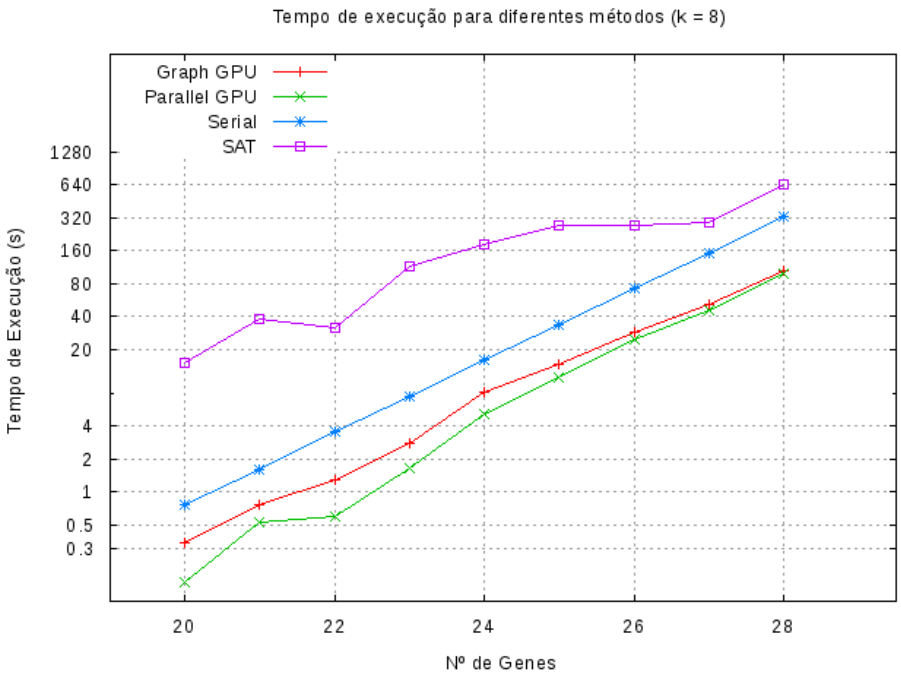


Figura 4.19: *Comparação entre os diferentes métodos ($k = 8$).*

Desempenho do método *Parallel GPU*

Nos experimentos realizados os métodos que mostram menor tempo de execução são *SAT* e *Parallel GPU*. O primeiro, segundo os autores (Dubrova e Teslenko [2011]), consegue manipular redes com mais de 1000 genes, identificando os atratores em tempo consideravelmente pequeno, com limite de 1200 segundos, para redes com grau médio de entrada baixo e limitado a 2, no máximo. Ainda, segundo o artigo, foram realizados experimentos com redes aleatórias com até 7000 genes e para várias destas redes, no tempo especificado, chegou-se a um resultado.

O método *Graph GPU* é limitado pela quantidade de memória disponível (memória da GPU), visto que o grafo de sequência de estados precisa estar totalmente carregado em memória, com isso, para o *hardware* utilizado nos experimentos, temos um limite para processar redes com até 28 genes, representados como número inteiros, ou seja $1GB$ de memória. Este limite, motivou a implementação do método *Parallel GPU* o qual consegue manipular redes de até 35 genes em tempo considerável - com limite de tempo de 1200 segundos - identificando os atratores e também o tamanho das bacias de atração, avaliando todo o espaço de estados. É possível parametrizar a execução, para que seja respeitado qualquer limite de tempo, obtendo-se apenas uma amostragem, que contém os atratores e tamanho das bacias de atração obtidos até o momento. Com a utilização de múltiplas GPUs o tempo total de execução também pode ser reduzido, sendo que se utilizadas duas ou mais GPUs idênticas, é possível obter um crescimento no desempenho praticamente linear. Nos experimentos realizados utilizando as GPUs *GeForce GTX 470* e *GeForce GTX 275* obtivemos um ganho de aproximadamente 75% no tempo de execução. A seguir, veremos as comparações do tempo de execução do método *Parallel GPU* com relação a todos os outros métodos (*Serial*, *Graph GPU* e *SAT*), inclusive, mostrando a curva de desempenho ou *speedup*, calculado como:

$$speedup = \frac{Ti}{Tp},$$

onde, Ti é o tempo do método a ser comparado, isto é tempo para os métodos *Serial*, *Graph GPU* e *SAT*, e Tp é o tempo do método alvo da comparação, no caso, o método *Parallel GPU*. O tempo de execução das comparações são exibidos no mesmo gráfico e separadamente é exibido um gráfico com o *speedup*. Utilizamos escala logarítmica para melhor visualização dos dados.

Desempenho ($k = 2$)

Para $k = 2$, a Figura 4.20 mostra em conjunto as curvas de tempo de execução para os métodos *Serial* e *Parallel GPU*. Podemos ver que o método *Serial* está com tempo superior ao método *Parallel GPU* em aproximadamente 20×, mostrando que na média, o tempo de execução do método *Serial* é 20× maior. Isso é confirmado pelo gráfico da Figura 4.21, que mostra o ganho de desempenho obtido da a implementação em GPU com relação à implementação serial (na média, temos $speedup = 20$).

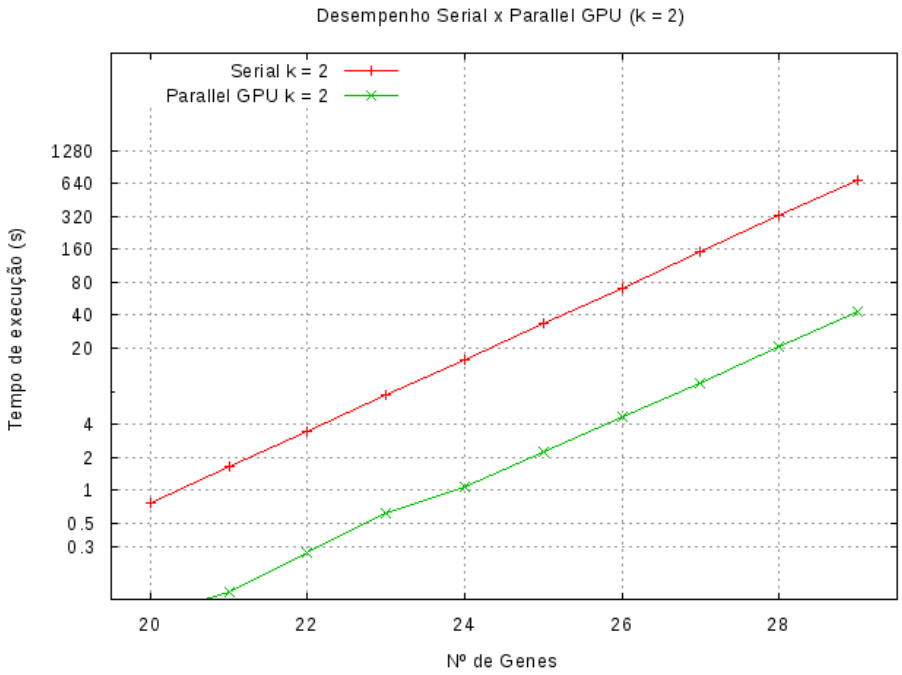


Figura 4.20: *Comparação do tempo de execução entre os métodos* Serial *e* Parallel GPU *para* $k = 2$.

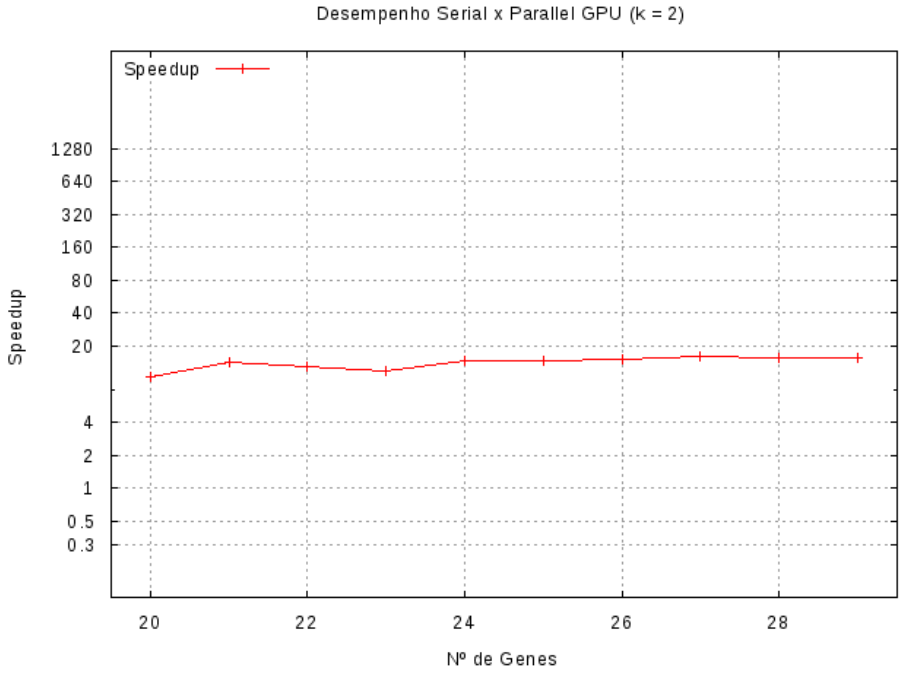


Figura 4.21: *Speedup* Parallel GPU *com relação ao método* Serial *para* $k = 2$.

A comparação com o método *Graph GPU*, exibido na Figura 4.22 mostra que o método *Parallel GPU* é aproximadamente 4× mais rápido que o mesmo, o que é visto na curva de desempenho da Figura 4.23.

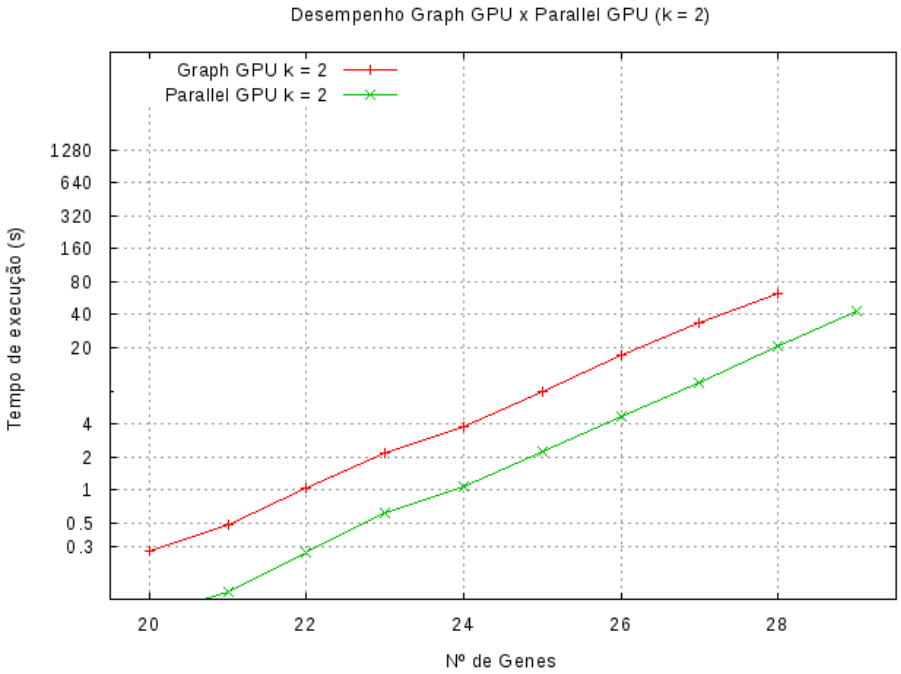


Figura 4.22: *Comparação do tempo de execução entre os métodos* Graph GPU *e* Parallel GPU *para* $k = 2$.

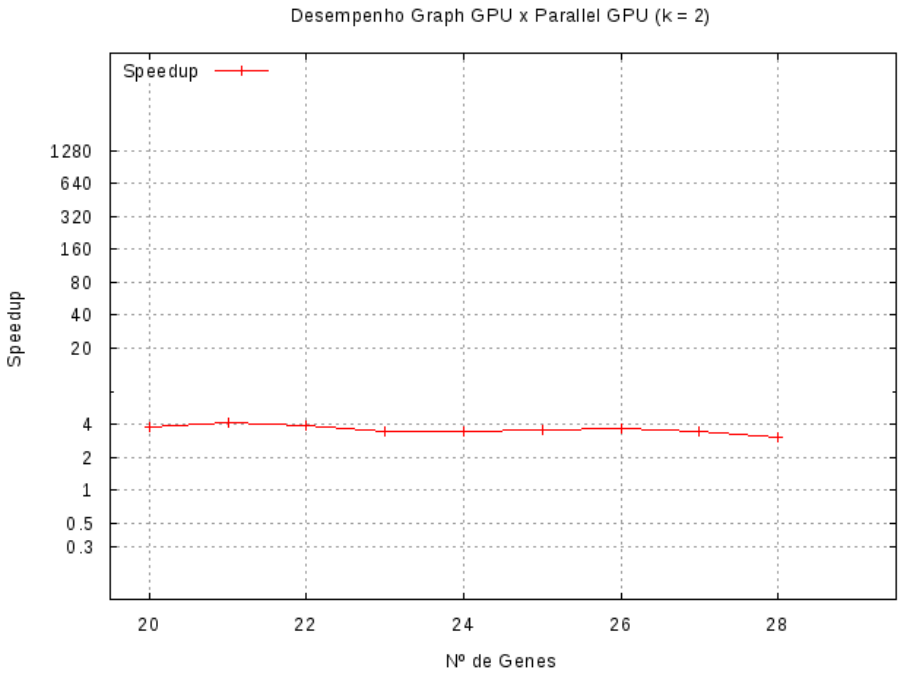


Figura 4.23: *Speedup* Parallel GPU *com relação ao método* Graph GPU *para* $k = 2$.

Para o método *SAT*, exibido na Figura 4.24, temos a mesma medição. Nesta figura podemos ver que o método *SAT*, para configuração $k = 2$, mostra tempo de execução menor que método *Parallel GPU*. Com isso o gráfico de desempenho, exibido na Figura 4.25 tem um curva na qual

o valor do *speedup* decresce com aumento do número de genes, mostrando que o método *Parallel GPU* é inferior em desempenho para esta configuração. Porém, vale ressaltar que o método *SAT* somente obtém os atratores (não contabiliza o tamanho das bacias de atração), mostrando ser um bom algoritmo para ser aplicado para grau médio de entrada baixo. Em seguida são mostrados os mesmos dados para $k = 6$ e $k = 8$.

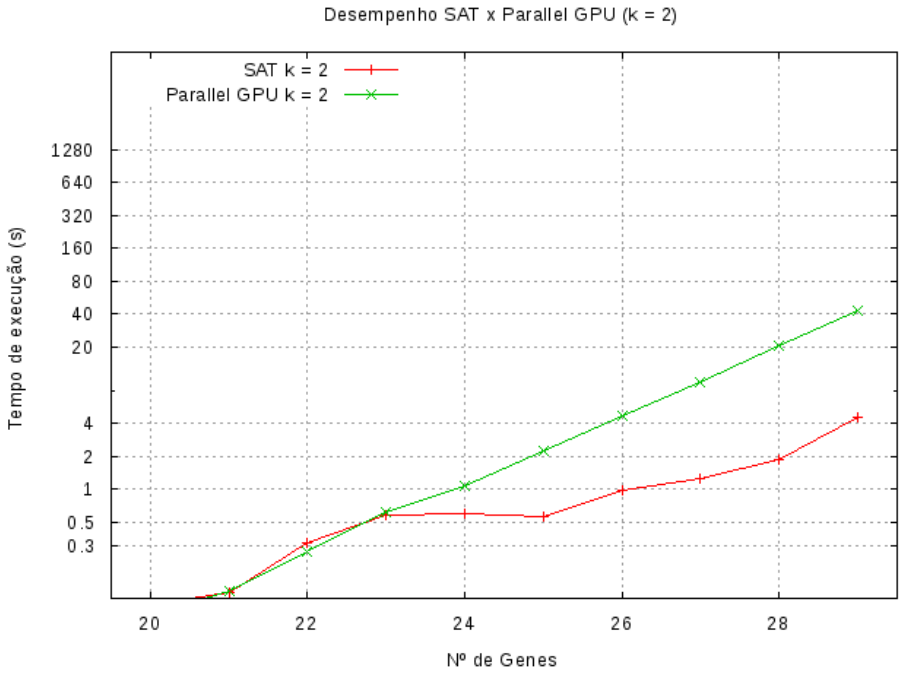


Figura 4.24: *Comparação do tempo de execução entre os métodos* SAT *e* Parallel GPU *para* $k = 2$.

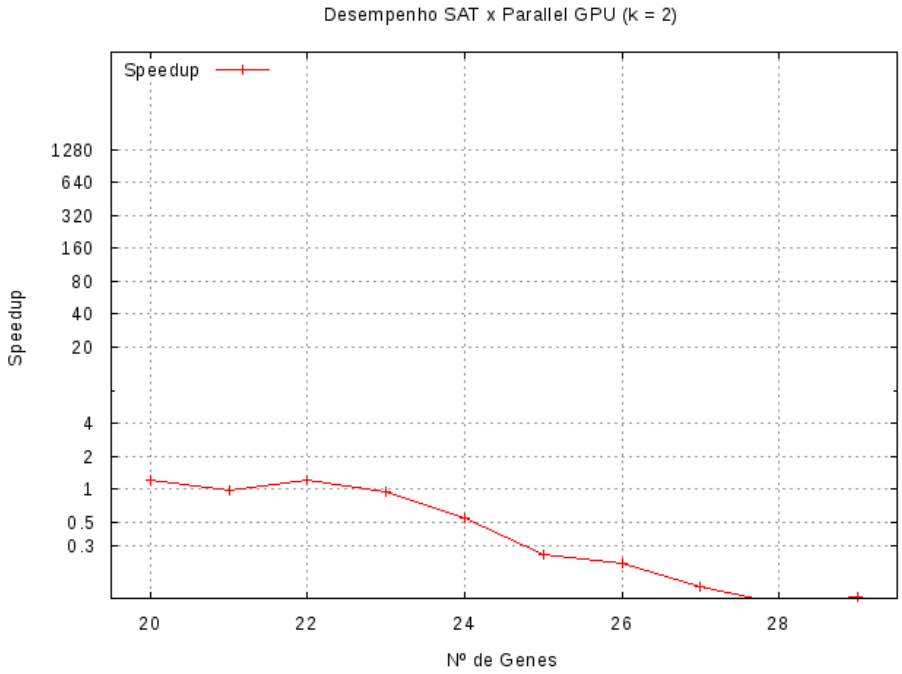


Figura 4.25: *Speedup* Parallel GPU *com relação ao método* SAT *para* $k = 2$.

Desempenho ($k = 6$)

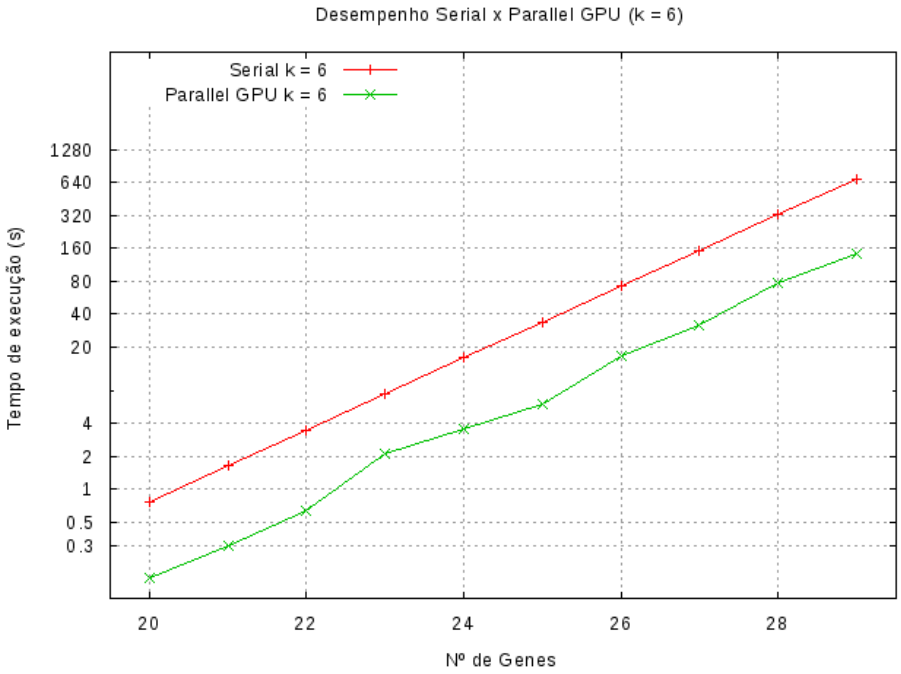


Figura 4.26: *Comparação do tempo de execução entre os métodos* Serial *e* Parallel GPU *para* $k = 6$.

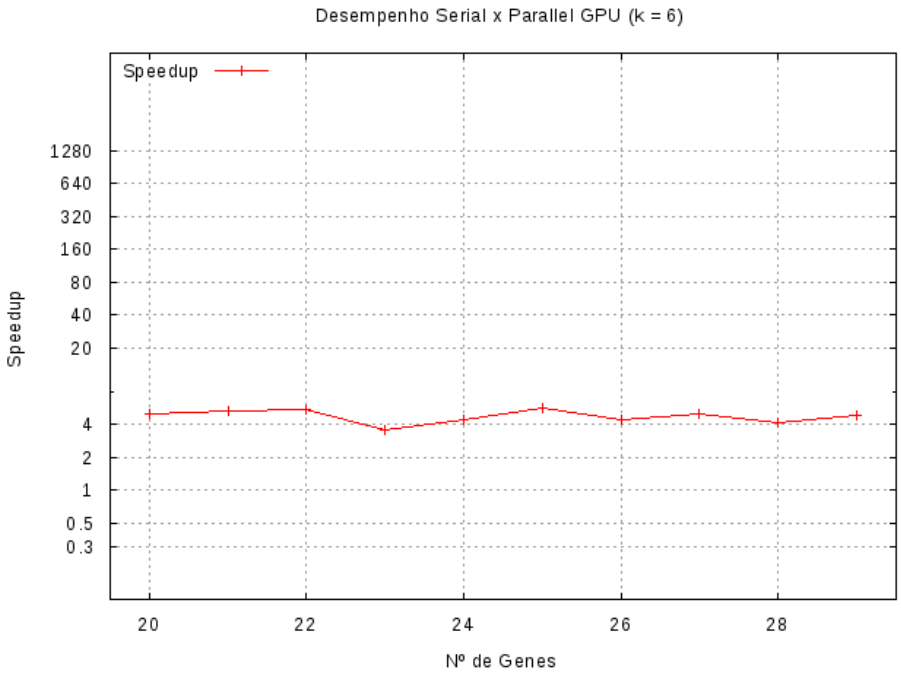


Figura 4.27: *Speedup* Parallel GPU *com relação ao método* Serial *para* $k = 6$.

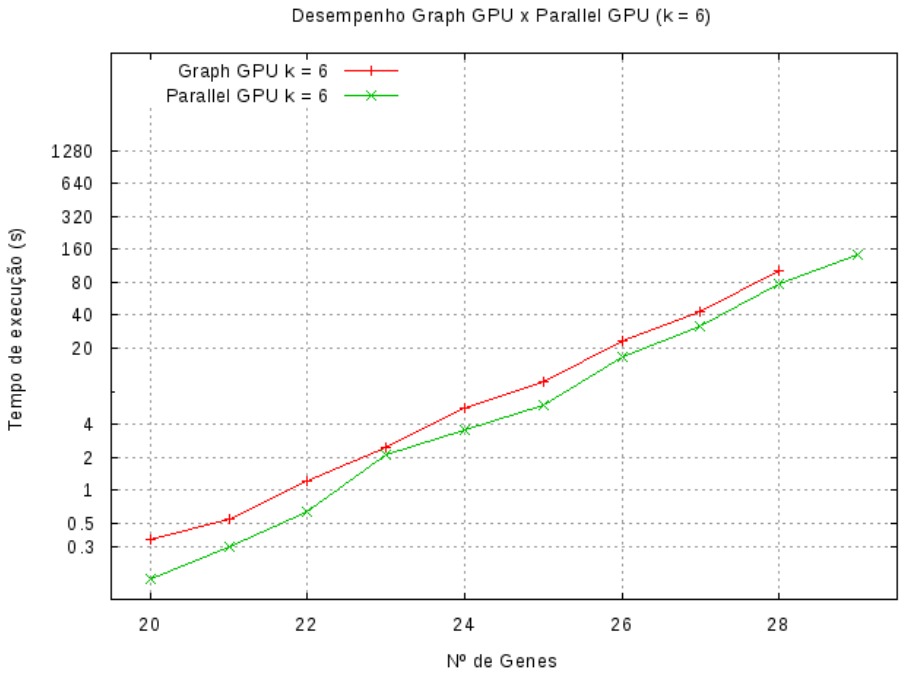


Figura 4.28: *Comparação do tempo de execução entre os métodos* Graph GPU *e* Parallel GPU *para* $k = 6$.

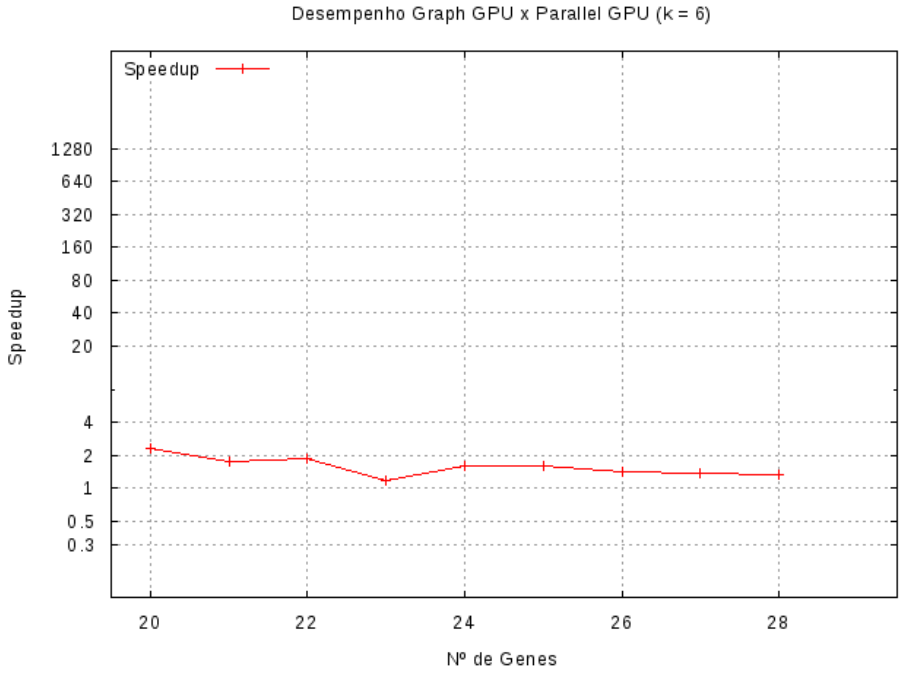


Figura 4.29: *Speedup* Parallel GPU *com relação ao método* Graph GPU *para* $k = 6$.

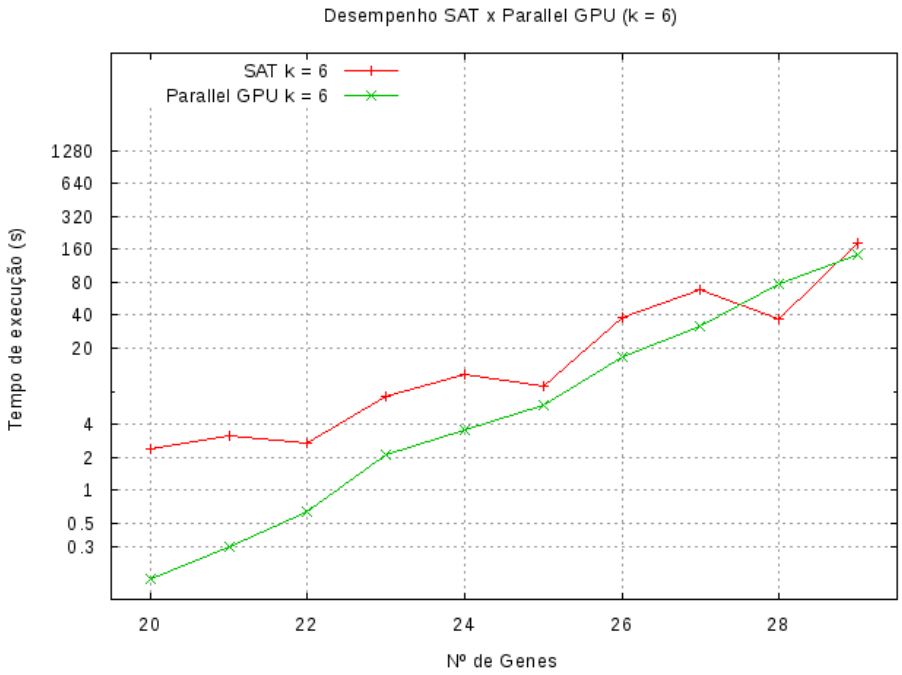


Figura 4.30: *Comparação do tempo de execução entre os métodos* SAT *e* Parallel GPU *para* $k = 6$.

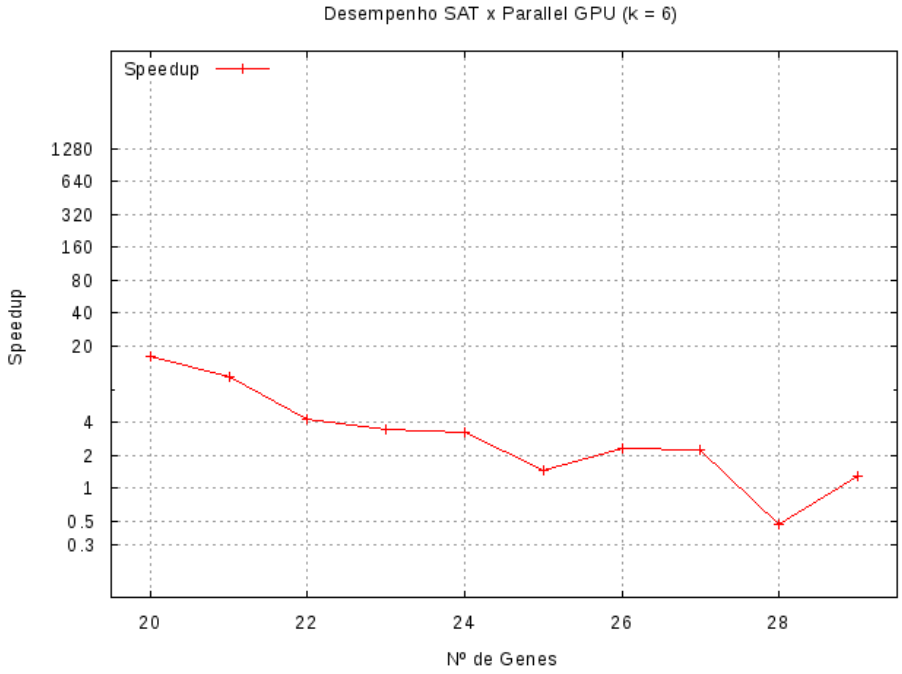


Figura 4.31: *Speedup* Parallel GPU *com relação ao método* SAT *para* $k = 6$.

Desempenho ($k = 8$)

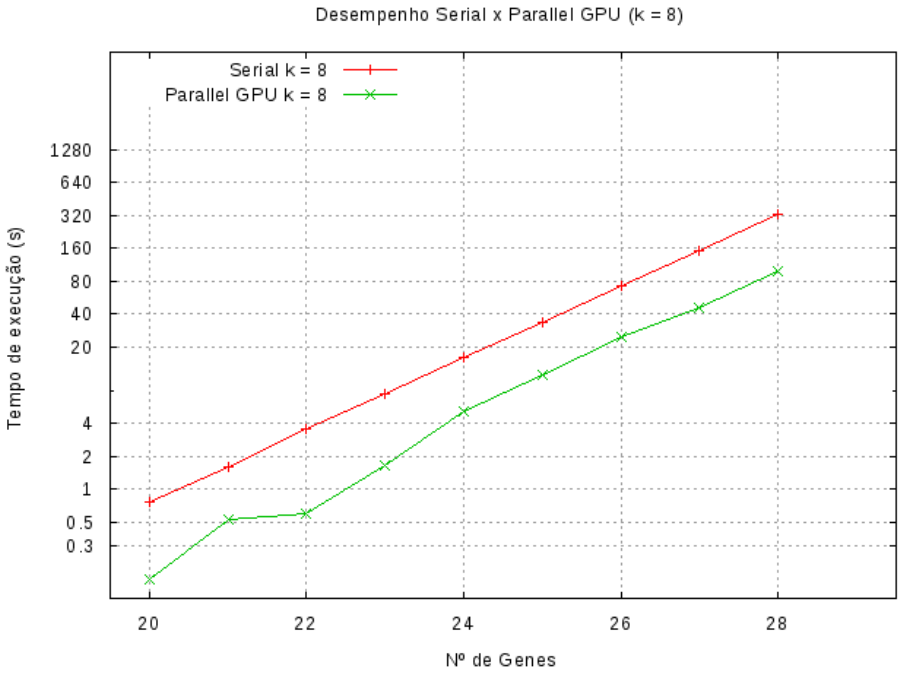


Figura 4.32: *Comparação do tempo de execução entre os métodos* Serial *e* Parallel GPU *para* $k = 8$.

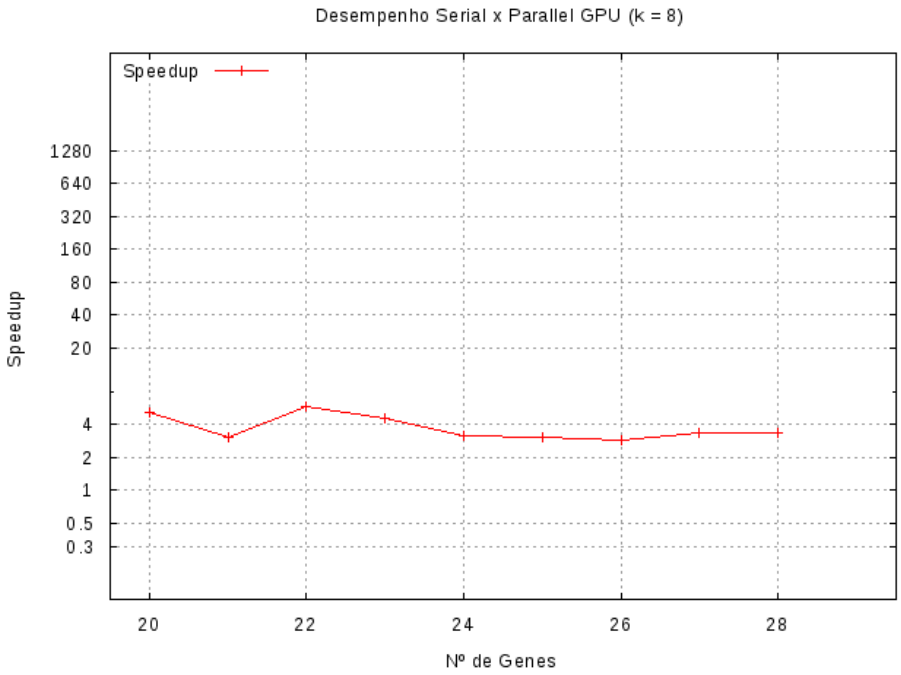


Figura 4.33: *Speedup* Parallel GPU *com relação ao método* Serial *para* $k = 8$.

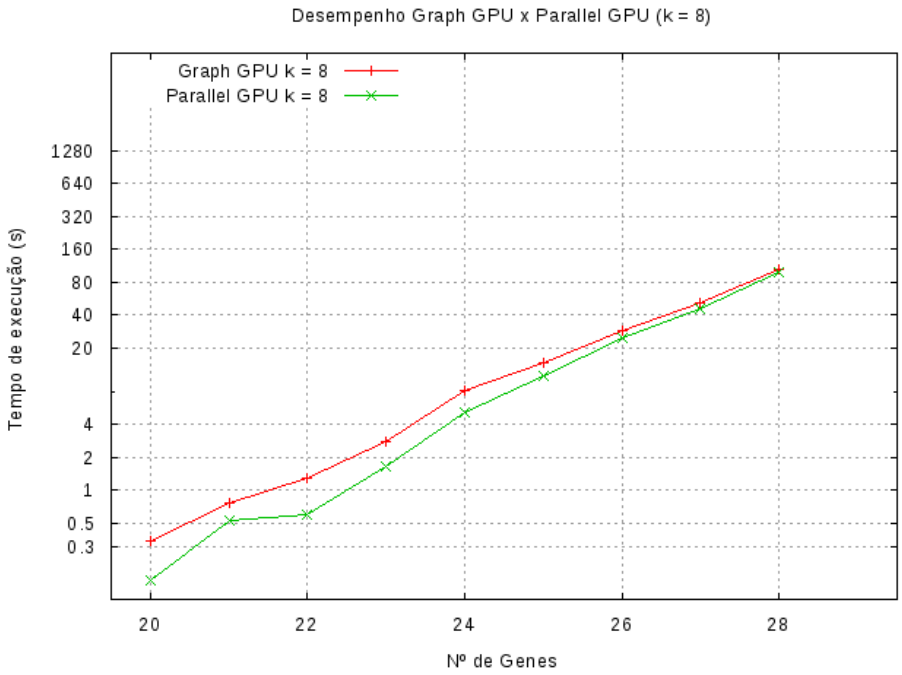


Figura 4.34: *Comparação do tempo de execução entre os métodos* Graph GPU *e* Parallel GPU *para* $k = 8$.

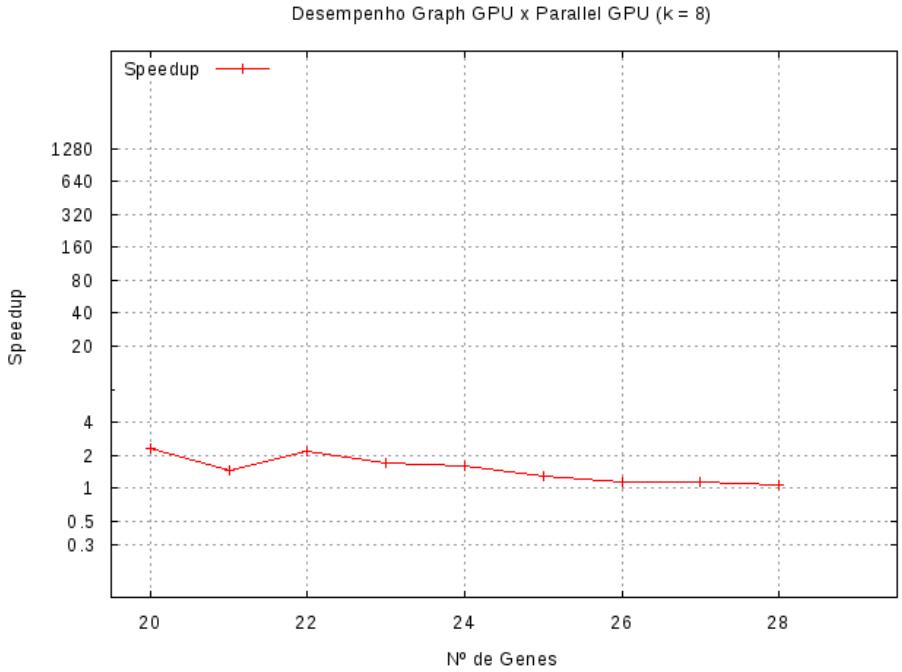


Figura 4.35: *Speedup* Parallel GPU *com relação ao método* Graph GPU *para* $k = 8$.

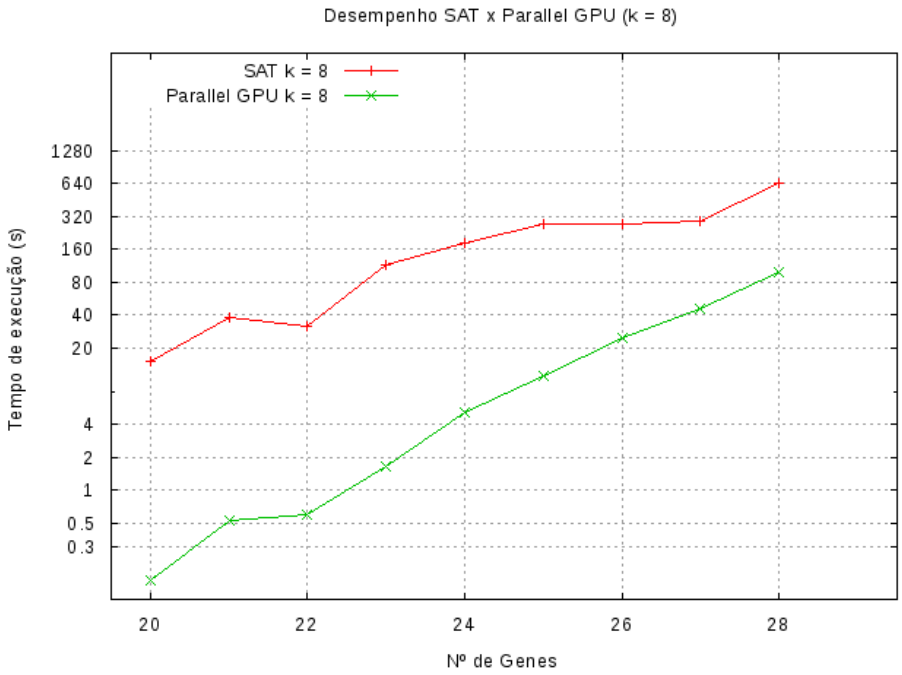


Figura 4.36: *Comparação do tempo de execução entre os métodos* SAT *e* Parallel GPU *para* $k = 8$.

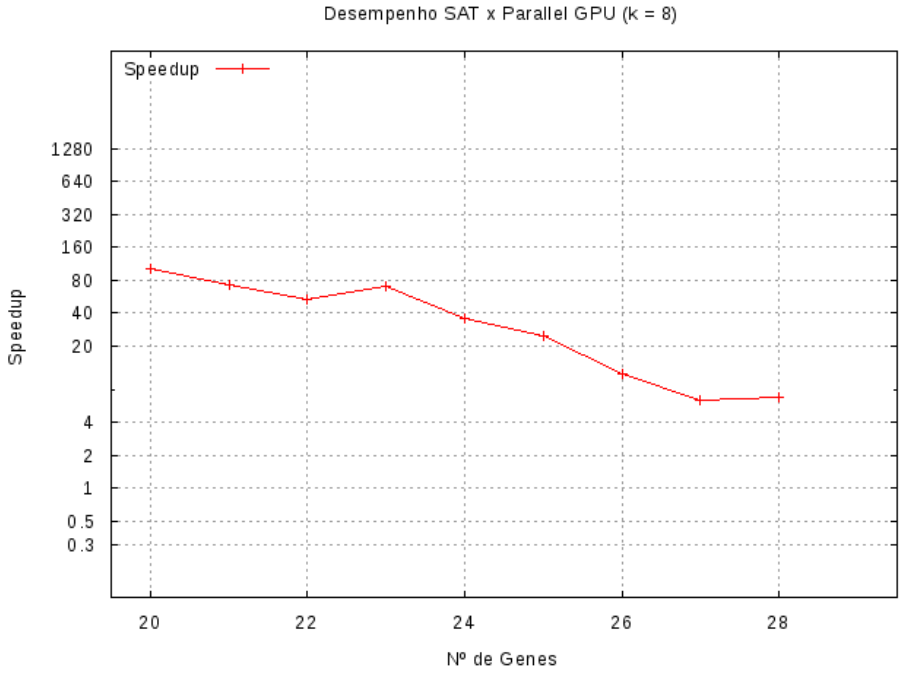


Figura 4.37: *Speedup* Parallel GPU *com relação ao método* SAT *para* $k = 8$.

Estatística das Redes Utilizadas

Nas figuras a seguir, são mostradas algumas informações, como número médio de atratores e tamanho médio da maior bacia de atração para as redes aleatórias utilizadas. Para utilização em outros trabalhos, por exemplo, em trabalhos que utilizam entropia para atingir algum objetivo, estas medições são essenciais. Na Figura 4.38, vemos que o grau de entrada influencia diretamente no número médio de atratores por rede. Por exemplo, para redes com 24 genes e grau médio de entrada $k = 2$, temos em média 960 atratores por rede. Com o aumento do grau máximo de entrada o número de atratores diminui consideravelmente, indo para 60 para $k = 6$ e 20 para $k = 8$.

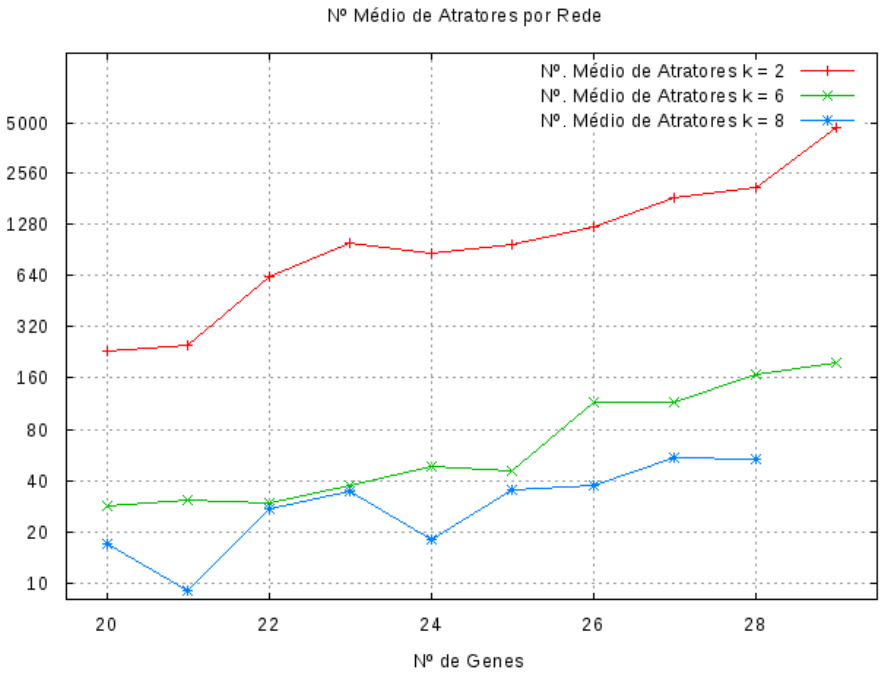


Figura 4.38: *Número médio de atratores. Redes aleatórias de grau de entrada máximo (k) igual a* 2, 6 *e* 8.

Na Figura 4.39, temos um histograma que mostra o tamanho das maiores bacias de atração. Para $k = 2$, como temos um número maior de atratores, o espaço de estados está disperso entre os atratores, sendo que as maiores bacias de atração ocupam em média uma taxa de 15% a 20% do espaço de estados. Para $k = 6$ temos de 35% a 40% e para $k = 8$ temos de 55% a 60%, mostrando uma proporção maior na quantidade de estados da maior bacia de atração nestas redes.

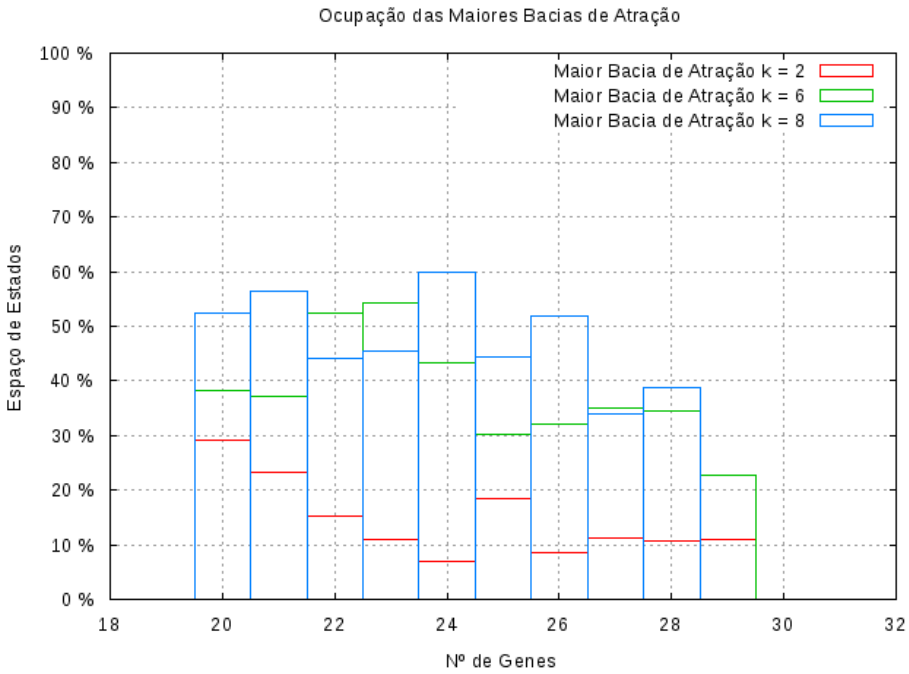


Figura 4.39: *Tamanho da maior bacia de atração. Redes aleatórias de grau de entrada máximo* $k = \{2, 6, 8\}$.

Capítulo 5

Considerações Finais

5.1 Conclusão

Neste trabalho estudamos o problema de identificação de atratores e tamanho das bacias de atração em redes booleanas, especificamente redes booleanas limiarizadas, propostas por Li *et al.* [2004]. O foco deste trabalho foi a implementação, utilizando métodos de alto desempenho e processamento GPU, de uma solução para o problema. O modelo de redes booleanas com limiar foi escolhido por ter destaque na modelagem de processos biológicos e também pela sua representação matricial, a qual simplificou a representação da rede na implementação em GPU. Foram desenvolvidos dois algoritmos, um que realiza contagem o tamanho das bacias de atração utilizando conceitos de grafos e um algoritmo de coloração de grafos implementado em GPU e outro que realiza o mesmo processo realizando trajetórias em paralelo. Uma das limitações na utilização do algoritmo de componentes conexas é a necessidade de manter o grafo em memória. Com pouco mais de 28 genes consome-se em torno de 1 GB de memória da GPU (pois é necessário o armazenamento de 2^{28} inteiros) e a tarefa torna-se inviável caso o número de genes aumente. Já o algoritmo de trajetórias em paralelo não carrega o grafo de sequência de estados em memória. Ao invés disso, apenas transita pela sequência de estados armazenando informações pertinentes até atingir o atrator. Uma desvantagem neste método é a possível repetição de trajetórias, o que poderia ser minimizado com a utilização de um *cache*. Esta análise é deixada como trabalho futuro.

Para efeito de comparação do tempo de execução, utilizamos a implementação padrão (geração do espaço de estados e caminhamento para frente até identificar os atratores e tamanho das bacias de atração - totalmente serial) e um método que parece promissor na identificação dos atratores. Este método é baseado no problema de satisfatibilidade booleana (SAT), foi desenvolvido no trabalho de Dubrova e Teslenko [2011]. Utilizamos a implementação disponibilizada e também fizemos uma adaptação do modelo de redes booleanas limiarizadas para atender ao padrão de entrada do necessário para execução do *software*. O problema tratado neste trabalho é de difícil resolução e com uso de GPUs conseguimos ganho de desempenho (de aproximadamente 20× no tempo de processamento), e ainda, com aumento de várias ordens no tamanho máximo da entrada com relação à implementação sequencial (máximo 29 genes na implementação sequencial para 35 genes na implementação em GPU - o que é relativo à 10× o espaço de estados processado). Porém, mesmo assim, o problema ainda torna-se intratável, pelo tempo de processamento para entrada com maior número de genes que este.

No geral a aplicação de computação paralela e uso de GPUs parece interessante (como de fato auxiliou outros trabalhos, reduzindo consideravelmente o tempo de processamento das simulações realizadas - como no trabalho de Andrade [2012]). Por fim, como uma solução parcial, a fim de caracterizar apenas uma porção do espaço de estados, utilizando taxas do tamanho das bacias de atração, ao invés do tamanho exato, é uma alternativa que pode ser explorada utilizando a implementação desenvolvida neste trabalho. Como protótipo, para experimentação do software

(visto que GPUs profissionais não estão disponíveis para todos os usuários) foi desenvolvida uma aplicação web, a qual pode ser instalada em um servidor que possua GPUs profissionais, a fim de permitir seu uso de forma transparente aos usuários. Futuramente esta aplicação pode ser expandida para permitir, por exemplo, a inferência de redes.

5.2 Trabalhos Futuros

O software desenvolvido neste trabalho pode ser expandido de forma a obter-se uma ferramenta de análise de redes booleanas completa que execute sobre a web. Com isso é possível centralizar o processo de solução, sendo possível até utilização de processamento distribuído na resolução do problema.

- Verificar o comportamento de outros resolvedores SAT, inclusive resolvedores utilizando GPUs (Meyer *et al.* [2010]);
- Utilizar os métodos SAT e o método de *Trajetórias em Paralelo utilizando GPU* em conjunto (o método de trajetórias em paralelo poderia ser utilizado para realizar estimativas do tamanho das bacias de atração. Enquanto que o método SAT pode determinar parcialmente ou completamente quais são os atratores).
- No método de *Trajetórias em Paralelo*, verificar se é possível evitar, mesmo que parcialmente a repetição de trajetórias (por exemplo, utilizando uma região de *cache* das trajetórias mais utilizadas) e com isso reduzir o tempo de processamento;
- Incrementar a ferramenta web para incluir funcionalidade de inferência de redes;
- Verificar possibilidade de uso de processamento distribuído, para realizar processamento em diversos nós (que tenham GPUs);
- Verificar o uso de algoritmos reversos como os de Wuensche [1998], utilizados a partir do método SAT. A ideia é obter os atratores com método SAT e após isso, aplicar uma função inversa para cada estado, sucessivamente até caracterizar completamente a bacia de atração.

Apêndice A

Conversão de Matriz de Regulação para Funções Booleanas

Para fins de comparação do tempo de execução e utilização da implementação baseada no problema SAT, proposta por Dubrova e Teslenko [2011], utilizando o modelo de rede booleana limiarizada, é necessário realizar uma conversão da matriz de regulação para um formato de funções booleanas, o qual é utilizado como entrada no programa. O formato utilizado é similar ao formato *Berkeley Logic Interchange Format* (BLIF) (ber [1992]), comumente utilizado em ferramentas de síntese e verificação de circuitos e redes booleanas. O formato do arquivo de entrada para a implementação baseada em SAT é o seguinte:

```
# Número de genes
.v <n:número de genes>

# Definição de cada gene
# Gene 0
.n <gene 0> <k:grau_entrada> <input 2> <input 3> <input 4> ... <input_k>

# Tabela verdade <gene 0>
# <input 2> <input 3> ... <input_k> <valor funcao>
0 0 0 ... 0 <valor funcao>
1 0 0 ... 0 <valor funcao>
...
1 1 1 ... 1 <valor funcao>

# Gene 1
.n <gene 1> <m:grau_entrada> <input 2> <input 3> ... <input m>

# Tabela verdade <gene 0>
0 0 ... 0 <valor funcao>
1 0 ... 0 <valor funcao>
...
1 1 ... 1 <valor funcao>

...

# Gene (n-1)
.n (n-1) <x:grau_entrada> <input 2> <input 3> ... <input x>
# Tabela verdade <gene (n-1)>
0 0 ... 0 <valor funcao>
1 0 ... 0 <valor funcao>
```

```
...
1 1 ... 1 <valor funcao>
```

Para realizar a conversão da matriz de regulação para o formato BLIF utilizado (descrito acima), é utilizado o seguinte algoritmo, o qual imprime matriz de regulação no formato especificado:

Algoritmo 8: Conversão da matriz de regulação para funções booleanas

Entrada: A (matriz de regulação gênica n x n), n (número de genes em questão).
Saída: Impressão da matriz de regulação no formato de funções booleanas.

```
1  imprime( ".v"+ " " + n + "\n") ;
2  para i de 0 até n faça
3      // Obtém o grau de entrada para o gene i
4      g = grauEntradaGene(i, A) ;
5      // Definição de cada gene
6      imprime( ".n "+ i + " " + g + " " ) ;
7      para cada input j do gene i faça
8          imprime( j + " " ) ;
9      fim
10     imprime( "\n") ;
11     // Tabela verdade
12     para cada "string binaria"k de 000...0 ate 111...1 de g bits faça
13         // Valor da função booleana para a string
               binaria k
14         f ← lim(dot(k, A)) ;
15         imprime(bin(k)) ;
16         imprime("" + f) ;
17         imprime( "\n") ;
18     fim
19 fim
```

Apêndice B

Aplicação web

A aplicação web permite o processamento de uma matriz de regulação em formato texto. Para isso o usuário deve carregar um arquivo contendo a descrição da matriz de regulação no seguinte formato:

$$
\begin{array}{ccccc}
n & & & & \\
a_{11} & a_{12} & a_{13} & \ldots & a_{1n} \\
a_{21} & a_{22} & a_{23} & \ldots & a_{2n} \\
a_{31} & a_{32} & a_{33} & \ldots & a_{3n} \\
\ldots & & & & \\
a_{n1} & a_{n2} & a_{n3} & \ldots & a_{nn}
\end{array}
$$

Onde n é o número de genes da rede e cada a_{ij} é o valor da regulação do gene i (coluna) para o gene j (linha), definido como:

$$
a_{ij} = \begin{cases} -1, & \text{para uma regulação negativa (inibição) do gene } i \text{ para } j \\ 1, & \text{para uma regulação positiva (ativação) do gene } i \text{ para } j \\ 0, & \text{caso contrário (não há influência) do gene } i \text{ para } j \end{cases}
$$

Em seguida são exibidas imagens do aplicativo web.

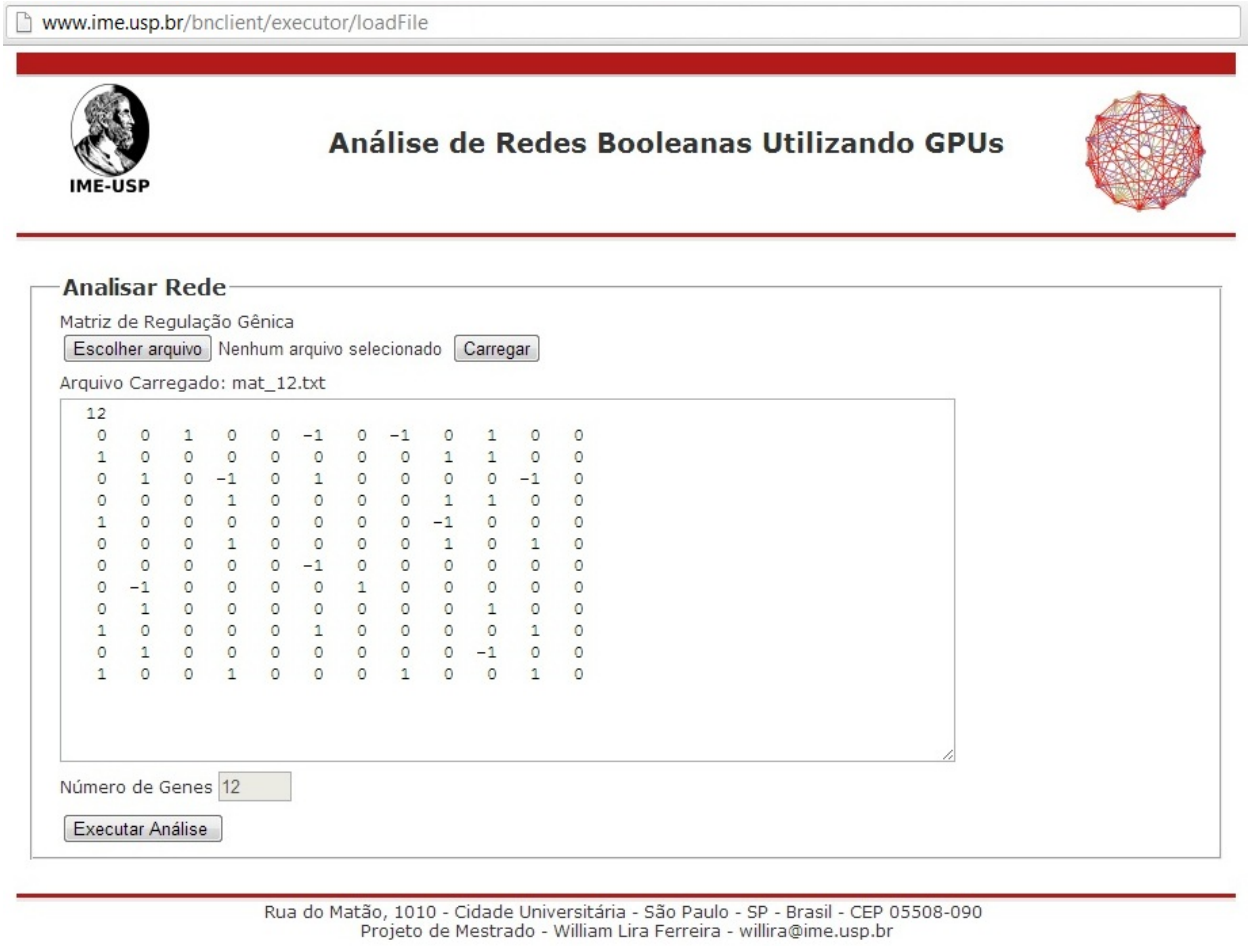


Figura B.1: *Aplicação web: carregamento de arquivo de matriz de regulação.*

Figura B.2: *Aplicação web: listagem da situação do processamento para uma execução não finalizada.*

Figura B.3: *Aplicação web: listagem da situação do processamento para execuções finalizadas.*

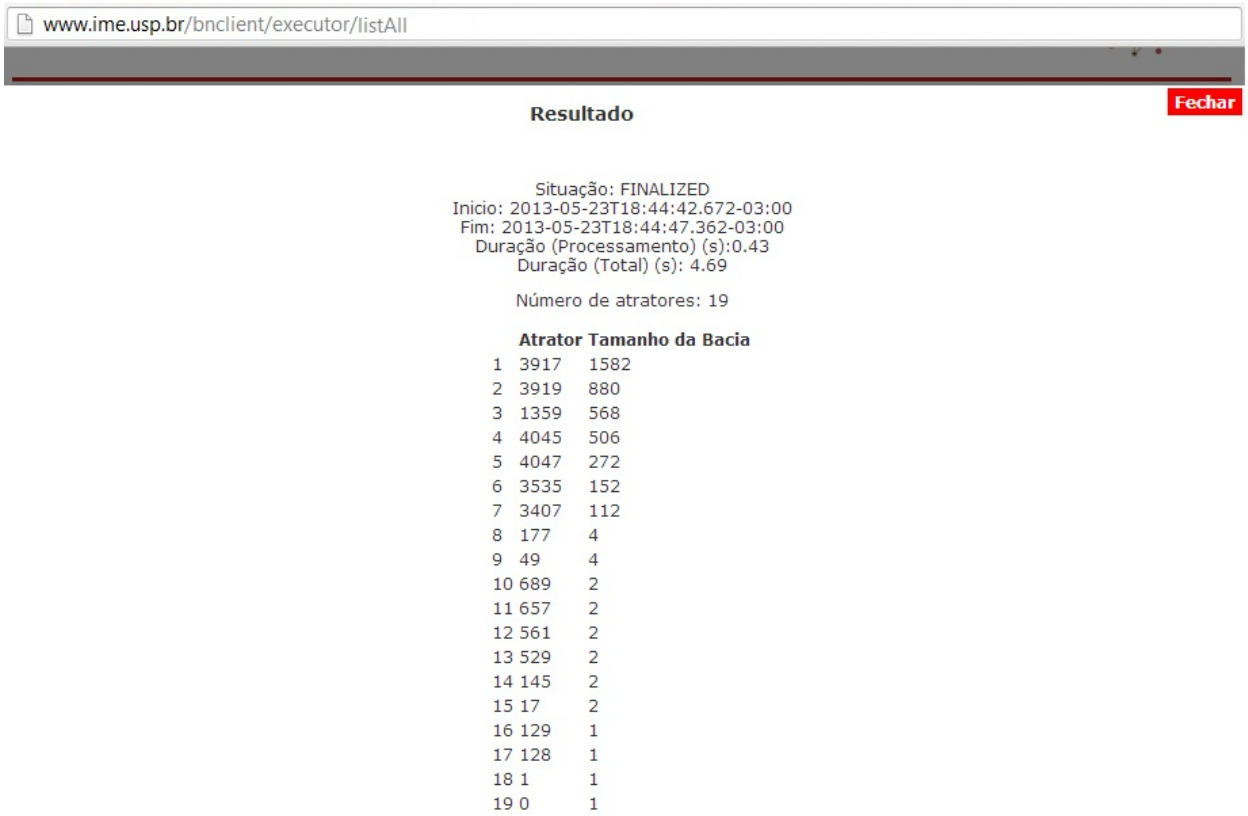


Figura B.4: *Aplicação web: resultado do processamento.*

Referências Bibliográficas

ber(1992) Berkeley logic interchange format (blif), Julho 1992. University of California. Citado na pág. 81

Akutsu *et al.***(2008)** Tatsuya Akutsu, Morihiro Hayashida e Takeyuki Tamura. Algorithms for inference, analysis and control of boolean networks. Em *Proceedings of the 3rd international conference on Algebraic Biology*, AB '08, páginas 1–15, Berlin, Heidelberg. Springer-Verlag. ISBN 978-3-540-85100-4. doi: 10.1007/978-3-540-85101-1_1. URL http://dx.doi.org/10.1007/978-3-540-85101-1_1. Citado na pág. 2, 9, 33

Andrade(2012) Tales Pinheiro de Andrade. Interações gênicas usando redes booleanas limiarizadas modeladas como um problema de satisfação de restrições, 2012. Dissertação de Mestrado, Instituto de Matemática e Estatística, Universidade de São Paulo, Brasil. Citado na pág. 2, 5, 29, 30, 31, 53, 79

Asanovic *et al.***(2006)** Krste Asanovic, Ras Bodik, Bryan Christopher Catanzaro, Joseph James Gebis, Parry Husbands, Kurt Keutzer, David A. Patterson, William Lester Plishker, John Shalf, Samuel Webb Williams e Katherine A. Yelick. The landscape of parallel computing research: A view from berkeley. Relatório Técnico UCB/EECS-2006-183, EECS Department, University of California, Berkeley. URL http://www.eecs.berkeley.edu/Pubs/TechRpts/2006/EECS-2006-183.html. Citado na pág. 16

Brown(2002) Terence A. Brown. *Genomes*. Oxford: Wiley-Liss, 2 edição. URL http://www.ncbi.nlm.nih.gov/books/NBK21128/. Citado na pág. 1

de Jong *et al.***(2003)** Hidde de Jong, Johannes Geiselmann, Céline Hernandez e Michel Page. Genetic network analyzer: qualitative simulation of genetic regulatory networks. *Bioinformatics*, 19(3): 336–344. Citado na pág. 2, 9

D'Haeseleer *et al.***(1999)** P. D'Haeseleer, S. Liang e R. Somogyi. Gene expression data analysis and modeling. *Proceedings of the Pacific Symposium on Biocomputing*, páginas 1–34. Citado na pág. 1

Doran(2007) Robert W. Doran. The gray code. *Journal of Universal Computer Science*, 13(11):1573–1597. Citado na pág. 36

Dubrova e Teslenko(2011) Elena Dubrova e Maxim Teslenko. A sat-based algorithm for finding attractors in synchronous boolean networks. *IEEE/ACM Trans. Comput. Biol. Bioinformatics*, 8(5): 1393–1399. ISSN 1545-5963. doi: 10.1109/TCBB.2010.20. URL http://dx.doi.org/10.1109/TCBB.2010.20. Citado na pág. 2, 9, 13, 14, 29, 31, 67, 79, 81

Dubrova *et al.***(2005)** Elena Dubrova, Maxim Teslenko e Hannu Tenhunen. Computing attractors in complex dynamic networks. páginas 535–543. NQC. Citado na pág. 9, 33

Fauré *et al.***(2006)** Adrien Fauré, Aurélien Naldi, Claudine Chaouiya e Denis Thieffry. Dynamical analysis of a generic boolean model for the control of the mammalian cell cycle. *Bioinformatics*, 22(14):124–131. doi: 10.1093/bioinformatics/btl210. URL http://bioinformatics.oxfordjournals.org/content/22/14/e124.abstract. Citado na pág. 3

Fogel *et al.***(2007)** G.B. Fogel, D.W. Corne e Y. Pan. *Computational Intelligence in Bioinformatics*. IEEE Press Series on Computational Intelligence. John Wiley & Sons. ISBN 9780470199084. URL http://books.google.com.br/books?id=1aIbcQ1momQC. Citado na pág. 8

Free Software Foundation, Inc(2008) Free Software Foundation, Inc. The gnu mach reference manual. http://www.gnu.org/software/hurd/gnumach-doc/Inter-Process-Communication.html, Novembro 2008. Edição 0.4, Online, acessado em 27/05/2013. Citado na pág. 17

Friedman *et al.***(2000)** N. Friedman, L. Michal, I. Nachman e D. Pe'er. Using bayesian networks to analyze expression data. *Journal of Computational Biology*, 7(3-4):601–620. Citado na pág. 1, 5

Goodwin(1963) B. C. Goodwin. *Temporal Organization in Cells. A Dynamic Theory of Cellular Control Process*. Academic Press, New York. Citado na pág. 2, 5

Gray(1953) Frank Gray. Pulse code communication, Mar 1953. U.S. Patent 2,632,058. Citado na pág. 36

Hache *et al.***(2009)** H. Hache, H. Lehrach e R. Herwig. Reverse engineering of gene regulatory networks: A comparative study. *EURASIP Journal on Bioinformatics and Systems Biology*, 2009. doi:10.1155/2009/617281. Citado na pág. 6

Harish e Narayanan(2007) Pawan Harish e P. J. Narayanan. Accelerating large graph algorithms on the gpu using cuda. Em *Proceedings of the 14th international conference on High performance computing*, HiPC'07, páginas 197–208, Berlin, Heidelberg. Springer-Verlag. ISBN 3-540-77219-7, 978-3-540-77219-4. URL http://dl.acm.org/citation.cfm?id=1782174.1782200. Citado na pág. 25

Hawick *et al.***(2010)** K. A. Hawick, A. Leist e D. P. Playne. Parallel graph component labelling with gpus and cuda. *Parallel Comput.*, 36(12):655–678. ISSN 0167-8191. doi: 10.1016/j.parco.2010.07.002. URL http://dx.doi.org/10.1016/j.parco.2010.07.002. Citado na pág. 3, 35, 40

Hecker *et al.***(2008)** M. Hecker, S. Lambeck, S. Toepfer, E. van Someren e E. Guthke. Gene regulatory network inference: Data integration in dynamic models - a review. *BioSystems*, 96(2009):86–113. Citado na pág. 6

Higa *et al.***(2010)** C. H. A. Higa, V. H. P. Louzada e R. F. Hashimoto. Analysis of gene interactions using restricted boolean networks and time-series data. Em *ISBRA*, páginas 61–76. Citado na pág. 2

Higa *et al.***(2011)** C. H. A. Higa, V. H. P. Louzada, T. P. Andrade e R. F. Hashimoto. Constraint-based analysis of gene interactions using restricted boolean networks and time-series data. *ISBRA '10: 6th International Symposium on Bioinformatics Research and Applications*, páginas 1–18. Citado na pág. 2, 5, 8, 29

Hogeweg(2011) P. Hogeweg. The roots of bioinformatics in theoretical biology. 7(3). e1002021. doi:10.1371/journal.pcbi.1002021. Citado na pág. 1

Hogeweg(1978) P. Hogeweg. Simulating the growth of cellular forms. 31:90–96. Citado na pág. 1

Hopfensitz *et al.***(2013)** Martin Hopfensitz, Christoph Müssel, Markus Maucher e HansA. Kestler. Attractors in boolean networks: a tutorial. *Computational Statistics*, 28(1):19–36. ISSN 0943-4062. doi: 10.1007/s00180-012-0324-2. URL http://dx.doi.org/10.1007/s00180-012-0324-2. Citado na pág. 13

Irons(2006) David James Irons. Improving the efficiency of attractor cycle identification in boolean networks. *Physica D*, 217(1):7–21. URL http://linkinghub.elsevier.com/retrieve/pii/S016727890600090X. Citado na pág. 2, 9, 13

Karlebach e Shamir(2008) Guy Karlebach e Ron Shamir. Modelling and analysis of gene regulatory networks. *Nature Publishing Group*, 9(3):770–780. URL http://www.nature.com/nrm/journal/v9/n10/suppinfo/nrm2503_S1.html. Citado na pág. 2

Kauffman(1969) S. A. Kauffman. Metabolic stability and epigenesis in randomly constructed genetic nets. *Journal of Theoretical Biology*, 22(3):437–467. Citado na pág. 1, 5, 7, 8, 54

Kauffman(1993) S. A. Kauffman. *The origins of order: Self-organization and selection in evolution*. Oxford University Press. Citado na pág. 1, 2, 9

Khronos Group(2012) Khronos Group. Opencl 1.2 reference pages. http://www.khronos.org/opencl/, 2012. Online, accessed 13-March-2012. Citado na pág. 18

Khronos Group(2013) Khronos Group. Opencl 1.2 reference pages. http://openmp.org/, 2013. Online, acessado em 20/12/2012. Citado na pág. 47

Kirk e Hwu(2010) David B. Kirk e Wen mei Hwu. *Programming Massively Parallel Processors - A Hands-on Approach*. Elsevier. ISBN 978-0-12-381472-2. Citado na pág. 17, 18, 19, 21, 22, 26, 27

Li *et al.*(2004) F. Li, T. Long, Y. Lu, Q. Ouyang, C. Tang e C. Tang. The yeast cell-cycle network is robustly designed. páginas 4781–4786. Citado na pág. 2, 5, 9, 11, 29, 79

Lodish *et al.*(2000) H. Lodish, A. Berk, S.L. Zipursky, P. Matsudaira, D. Baltimore e J.E. Darnell. *Molecular Cell Biology*. W. H. Freeman, New York, USA. Citado na pág. 1

Meyer *et al.*(2010) Quirin Meyer, Fabian Schönfeld, Marc Stamminger e Rolf Wanka. 3-SAT on CUDA: Towards a massively parallel SAT solver. Em *Proc. High Performance Computing and Simulation Conference (HPSC)*, páginas 303–313. Citado na pág. 80

Müssel *et al.*(2010) Christoph Müssel, Martin Hopfensitz e Hans A. Kestler. Boolnet - an r package for generation, reconstruction and analysis of boolean networks. *Bioinformatics*, 26(10):1378–1380. doi: 10.1093/bioinformatics/btq124. URL http://bioinformatics.oxfordjournals.org/content/26/10/1378.abstract. Citado na pág. 2, 3, 9, 13

NVIDIA(2012a) NVIDIA. *CUDA API Reference Manual*. NVIDIA, January 2012a. versão 4.1. Citado na pág. 23

NVIDIA(2012b) NVIDIA. *CUDA C Best Practices Guide*. NVIDIA, January 2012b. versão 4.1. Citado na pág. 22, 24

NVIDIA(2010) NVIDIA. *The CUDA Compiler Driver NVCC*. NVIDIA, August 2010. versão 3.2. Citado na pág. 20

NVIDIA(2011) NVIDIA. *CUDA C Programming Guide*. NVIDIA, November 2011. versão 4.1. Citado na pág. 18, 19, 21, 23, 25, 26

NVIDIA(2007) NVIDIA. *GPU Computing Technical Brief*. NVIDIA, May 2007. versão 1.0.0. Citado na pág. 17

P. Hogeweg(1978) B. Hesper P. Hogeweg. Interactive instruction on population interactions. 8: 319–327. Citado na pág. 1

Pearl(1988) J. Pearl. *Probabilistic reasoning in intelligent systems: networks of plausible inference*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA. ISBN 0-934613-73-7. Citado na pág. 6

Sanders e Kandrot(2010) J. Sanders e E. Kandrot. *CUDA by Example - An introduction to General-Purpose GPU Programming*. Pearson Education. ISBN 978-0-13-138768-3. Citado na pág. 18

Santos(2012) Paulo Carlos Ferreira dos Santos. Ferramentas para extração de informações de desempenho em gpus nvidia. Dissertação de Mestrado, Instituto de Matemática e Estatística, Universidade de São Paulo, Brasil. Trabalho em andamento. Citado na pág. 21

Schena *et al.***(1995)** M. Schena, D. Shalon, R. W. Davis e P. O. Brown. Quantitative monitoring of gene expression patterns with a complementary dna microarray. *Science*, 270(5235):467–470. Citado na pág. 6

Shalon *et al.***(1996)** D. Shalon, S. J. Smith e P. O. Brown. A dna microarray system for analyzing complex dna samples using two-color fuorescent probe hybridization. *Genome Research*, 7(6): 639–645. Citado na pág. 6

Shmulevich *et al.***(2002)** I. Shmulevich, E. R. Dougherty e W. Zhang. From boolean to probabilistic boolean networks as models of genetic regulatory networks. *Proceedings of the IEEE*, 90(11): 1778–1792. Citado na pág. 1, 5

Shmulevich *et al.***(2005)** I. Shmulevich, E. R. Dougherty, J. Chen e Z. J. Wang. *Genomic Signal Processing and Statistics*. EURASIP Book Series on Signal Processing and Communications. ISBN 977-5945-07-0. Citado na pág. 2

Silberschatz e Peterson(1994) A. Silberschatz e J. Peterson. *Operating Systems Concepts*. Addison-Wesley. Citado na pág. 16

Skodawessely e Klemm(2011) Thomas Skodawessely e Konstantin Klemm. Finding attractors in asynchronous boolean dynamics. páginas 439–449. Citado na pág. 2, 9, 13, 33

Tanenbaum(1992) A. S. Tanenbaum. *Modern Operating Systems*. Prentice-Hall. Citado na pág. 16

Wahde e Hertz(2000) Mattias Wahde e John Hertz. Coarse-grained reverse engineering of genetic regulatory networks. *Biosystems*, 55:129–136. Citado na pág. 2, 7

Wikipedia(2013) Wikipedia. Thread. https://en.wikipedia.org/wiki/Thread_(computing), Maio 2013. Online, acessado em 27/05/2013. Citado na pág. 17

Wuensche(1998) Andrew Wuensche. Discrete dynamical networks and their attractor basins. Em *Complexity International*, páginas 3–21. Citado na pág. 2, 9, 10, 13, 80

Zhang *et al.***(2007)** S.Q. Zhang, W.K. Ching, M.K. Ng e T. Akutsu. Simulation study in probabilistic boolean network models for genetic regulatory networks. *International Journal on Data Mining and Bioinformatics*, 1(3):217–240. Citado na pág. 5