Problema do Fluxo Máximo em Redes Utilizando OpenMP e CUDA

Luiz Fernando Alvino

SERVIÇO DE PÓS-GRADUAÇÃO DA FACOM-UFMS

Data de Depósito:

Assinatura:______________________________

Luiz Fernando Alvino

Orientador: *Prof. Dr. Marco Aurélio Stefanes*

Dissertação apresentada ao curso de Pós Graduação em Ciências da Computação, da Universidade Federal de Mato Grosso do Sul, como requisito parcial à obtenção do título de Mestre em Ciência da Computação.

FACOM - Universidade Federal de Mato Grosso do Sul.
Setembro/2015

Ao Pai Celestial em primeiro lugar,

À minha esposa,
Ana Paula,

Aos meus filhos,
Matheus e Augusto,

Ao meu orientador,
Prof. Dr. Marco Aurélio Stefanes.

Agradecimentos

- Agradeço ao Pai Celestial, por sempre estar ao meu lado, pela força e inspiração nos momentos mais difíceis.
- À Ana Paula, minha querida esposa, pelo amor, apoio e paciência.
- Ao Matheus e Augusto, meus filhos, por encherem minha vida de alegria e realização.
- Ao Professor Marco Aurélio Estefanes pela orientação, guia e compreensão.
- À toda minha família e amigos pelos conselhos, apoio e incentivo.
- Aos amigos do CTEI-UFMS, Valter, Jean e Ângelo, pelas experiências trocadas e pelo companheirismo.
- Às instituições IFMS e UFMS, por me darem oportunidades e me auxiliarem durante esta jornada.

Resumo

O problema do fluxo máximo em redes é um problema fundamental de teoria dos grafos, com muitas aplicações importantes. Os algoritmos para o fluxo máximo baseados no método *push-relabel* são conhecidos por serem mais eficientes assintoticamente e terem menor tempo de execução na prática. Vários algoritmos paralelos foram propostos, mas poucos deles tiveram tempos de execução menores do que a implementação *hipr* de Goldberg, baseada em *push-relabel*. O objetivo geral desta dissertação é discutir as soluções sequenciais e paralelas para o problema do fluxo máximo em redes. Uma contribuição relevante é que propomos um novo algoritmo paralelo híbrido OpenMP-CUDA que explora a paralelização das heurísticas rotulação global e rotulação *gap*, além de utilizar o processamento em CPU e GPU adaptativamente para maximizar a eficiência de execução. Os resultados dos testes realizados mostram que esse algoritmo é até 5 vezes mais rápido do que a implementação *hipr*.

Abstract

The maximum flow problem is a fundamental problem of graph theory with important applications. Max-flow algorithms based on the push-relabel method are known to have better complexity bounds and faster practical execution. Other algorithms were proposed, but few had better execution speed than the best serial implementation, the Goldberg's *hipr*. The goal of this dissertation is to discuss the sequential and parallel solutions to the max-flow problem. A significant contribution is that we propose a new parallel hybrid algorithm OpenMP-CUDA that explores the parallelization of heuristics, such as global relabeling and gap relabeling, and use the processing in CPU and GPU adaptively to maximize execution efficiency. The results of the tests show that this algorithm is up to 5 times faster than the *hipr* implementation.

Sumário

Sumário xiv
Lista de Figuras xv
Lista de Tabelas xvii
Lista de Abreviaturas xix
Lista de Algoritmos xxi

1 Introdução **1**
1.1 Trabalhos relacionados 2
1.2 Objetivo 3

2 Arquiteturas paralelas **5**
2.1 Modelo PRAM 5
2.2 Memória Distribuída com MPI 6
2.3 Memória compartilhada com OpenMP 7
2.4 CUDA 10

3 Algoritmos paralelos para o problema do Fluxo Máximo em Redes **13**
3.1 Algoritmos baseados em Multi-threaded 13
3.2 Algoritmos baseados em CUDA 18
3.2.1 Push-relabel 18
3.2.2 Autoestabilizante 22
3.3 Algoritmos baseados em PRAM 25
3.4 Algoritmos baseados em MPI 25

4 Algoritmo híbrido OpenMP-GPU **29**
4.1 Algoritmo híbrido OpenMP-GPU 29
4.1.1 Operação *pushrelabel-gpu* 30
4.1.2 Sobreposição de *kernels* em CUDA 31
4.1.3 Utilização de *loop unrolling* 32
4.1.4 Paralelização da rotulação global utilizando OpenMP 34
4.1.5 Paralelização da rotulação *gap* utilizando OpenMP 34

4.1.6 Outros detalhes de implementação 35
4.2 Resultados computacionais 36

5 Conclusão **43**

Lista de Figuras

1.1 Exemplo de um fluxo em G. 3

2.1 O modelo de programação *fork-join* suportado pelo OpenMP. . . . 8
2.2 Por dentro de um SM. 11

3.1 Formação dos rótulos de distância inválidos: Na figura (a), o fluxo é empurrado em (a,b) enquanto b está sendo rotulado; e na figura (b) a aresta reversa (b,a) com $h(b) > h(a)+1$ é formada. 17
3.2 Um exemplo de particionamento de rede. 25
3.3 O modo rotular os vértices de fronteira. 27

4.1 Um exemplo de grafo do tipo Acyclic Dense. 36
4.2 Um exemplo de grafo do tipo Washington-RLG. 37
4.3 Um exemplo de grafo do tipo Genrmf. 38
4.4 Resultados computacionais em grafos Acyclic-Dense. 39
4.5 Resultados computacionais em grafos Genrmf-long. 39
4.6 Resultados computacionais em grafos Genrmf-wide. 40
4.7 Resultados computacionais em grafos Washington-RLG-long. . . 41
4.8 Resultados computacionais em grafos Washington-RLG-wide. . . 41

Lista de Tabelas

4.1 Tempos(s) e *speedups* para grafos Acyclic-Dense. 38
4.2 Tempos(s) e *speedups* para grafos Genrmf-long. 39
4.3 Tempos(s) e *speedups* para grafos Genrmf-wide. 40
4.4 Tempos(s) e *speedups* para grafos Washington-RLG-long. 40
4.5 Tempos(s) e *speedups* para grafos Washington-RLG-wide. 41

Lista de Abreviaturas

PRAM Parallel Random-Access Machine

SIMD Single Instruction Multiple Data

EREW Exclusive Read Exclusive Write

CREW Concurrent Read Exclusive Write

CRCW Concurrent Read Concurrent Write

MPI Message Passing Interface

CUDA Compute Unified Device Architecture

SM Symmetric Multiprocessor

SMP Symmetric Multiprocessing

SP Streaming Processor

DRAM Dynamic Random-Access Memory

FIFO First In First Out

TBB Treading Building Blocks

ARG Rotulação Global Assíncrona

XMT Explicit Multi-Threading

DIMACS Center for Discrete Mathematics and Theoretical Computer Science

GIS Geographic Information Systems

OpenMP Open Multi-Processing

ARB Architecture Review Board

GPU Graphics Processor Unit

Lista de Algoritmos

1 Algoritmo *push-relabel multi-threaded* livre de *lock* de Hong 15
2 Funções *init* e *push-relabel-cpu* do algoritmo CPU-GPU-Híbrido . 19
3 Algoritmo *push-relabel* CPU-GPU-Híbrido dinamicamente ajustado 20
4 Inicialização do algoritmo CPU-GPU-Híbrido 20
5 Função push-relabel-kernel 21
6 Rotulação global para CPU-GPU-Híbrido 22
7 Reduce_InFlow 24
8 Reduce_OutFlow 24
9 Algoritmo híbrido OpenMP-GPU 30
10 Descrição da função initialize 31
11 Descrição da função pushrelabel-cpu 32
12 Descrição da função pushrelabel-gpu 33
13 Rotulação global paralela 35
14 Rotulação gap paralela 35

CAPÍTULO 1

Introdução

O problema do fluxo máximo em redes é um problema clássico de teoria dos grafos com muitas aplicações importantes. Fluxo em redes pode ser utilizado para modelar sistemas de encanamento para abastecimento de água, sistemas de abastecimento de mercadorias em um conjunto de localidades, ou até mesmo escalas de tripulantes em uma companhia aérea. Existem aplicações em outras áreas, tal como visão computacional e geoprocessamento.

Muitos exemplos de aplicações do fluxo máximo em redes são encontrados no dia a dia. Um exemplo de aplicação do fluxo máximo em redes é maximizar o fluxo de uma rede de distribuição de suprimentos de uma companhia a partir de suas fábricas até chegar nos clientes. Outra aplicação é maximizar o fluxo de água partindo de um reservatório, passando por um sistema de aquedutos, até chegar às casas. Além disso, o fluxo máximo em redes pode ser aplicado a agendamento de recursos de transporte ou sistemas de informação geográfica.

Chamamos de rede um grafo orientado, ou seja, um conjunto de pontos, que são os vértices, interligados por arestas orientadas. A cada uma dessas arestas é atribuído um valor inteiro denominado capacidade da aresta. Esta capacidade é usada para representar restrições de transferência de produto. Podemos definir fluxo como sendo o montante de algum produto que é transferido de uma fonte a um destino.

Seja $G(V,E)$ um grafo orientado, onde a aresta $(u,v) \in E$ tem capacidade $c(u,v)$. Para conveniência de notação, $c(u,v)$ é definido como 0, se $(u,v) \notin E$. G tem um vértice fonte $s \in V$ e um vértice sorvedouro $t \in V$. $F(V,E,c,s,t)$ é chamado de fluxo em rede. Um fluxo em G é uma função de valor real f definida sobre $V \times V$ que satisfaz as seguintes restrições:

1. $f(u,v) \leq c(u,v)$, for $u,v \in V$.
2. $f(v,u) = -f(u,v)$, for $u,v \in V$.
3. $\sum_{v \in V} f(v,u) = 0$, for $u \in V - \{s,t\}$

O valor de um fluxo f é definido como $|f| = \sum_{u \in V} f(s,u)$, ou seja, o valor de um fluxo é a quantidade de fluxo de rede enviado de s para t. O problema do fluxo máximo busca um fluxo com o valor máximo. A Figura 1.1 mostra um exemplo de fluxo em G, sendo que o fluxo máximo neste caso é 5.

A capacidade residual de uma aresta $(u,v) \in E$ é $c_f(u,v) = c(u,v) - f(u,v)$. Uma aresta, pela qual $c_f(u,v) > 0$, é chamada de aresta residual. O conjunto de todas as arestas residuais em G é denotada por E_f. O grafo $G_f(V_f, E_f)$ é chamado de grafo residual.

1.1 Trabalhos relacionados

Algoritmos sequenciais e paralelos vêm sendo estudados para este problema. As primeiras soluções para o problema de fluxo máximo em redes são baseadas no método dos caminhos aumentantes de Ford e Fulkerson [10], que foi posteriormente melhorado através da escolha cuidadosa da ordem na qual os caminhos aumentantes são selecionados. Por exemplo, com o algoritmo $O(|V||E|^2)$ de Edmonds e Karp [9] e o algoritmo $O(|V|^2|E|)$ de Dinic [8]. O conceito de pré-fluxo foi introduzido por Karzanov [21], que leva a um algoritmo $O(|V|^3)$. Goldberg e Tarjan projetaram um método *push-relabel* [12] com $O(|V|^2|E|)$ operações e posteriormente melhoraram a complexidade através de estruturas de dados especiais [11].

Além desses algoritmos sequenciais, algoritmos paralelos também receberam muita atenção. Por exemplo, o algoritmo paralelo atribuído a Shiloach e Vishkin [23] executa em tempo $O(|V|^2 \log |V|)$ usando uma PRAM com $|V|$-processadores. Posteriormente Caragea e Vishkin [3] implementaram este algoritmo em uma arquitetura multi-core inspirada em PRAM.

Implementações práticas de algoritmos paralelos também foram investigadas intensivamente. Anderson e Setubal [1] ampliaram o algoritmo *push-relabel* com uma operação de *rotulação global* em CPU. Bader e Sachdeva [2] projetaram uma implementação paralela utilizando a heurística de rotulação *gap* com considerações no desempenho do *cache* no algoritmo *push-relabel*. Essas implementações paralelas, entretanto, compartilham uma funcionalidade comum de utilizar *locks* para proteger cada operação *push* e *relabel* em sua totalidade, que essencialmente serializam as operações se um vértice em comum está envolvido. Hong e He [15] projetaram um algoritmo paralelo assíncrono livre de *locks* em CPU e uma versão desse algoritmo baseada em uma

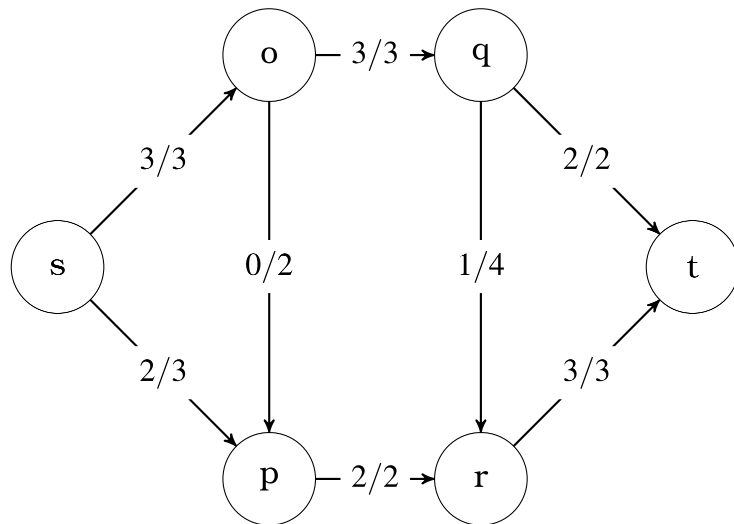


Figura 1.1: Exemplo de um fluxo em G.

arquitetura híbrida CPU-GPU [16], relatando *speedups* de até 2,5 vezes sobre o código sequencial mais rápido.

1.2 Objetivo

O objetivo deste trabalho é estudar e definir um algoritmo paralelo para o problema do fluxo máximo em redes que utilize uma arquitetura híbrida em CUDA e OpenMP. Baseando-se nas abordagens existentes, desenvolver uma implementação que una as duas arquiteturas e consiga ganhos de *speedup*.

Este trabalho está organizado da seguinte forma: no capítulo 2 são descritas três arquiteturas paralelas existentes. O capítulo 3 apresenta as soluções paralelas existentes para essas arquiteturas. No capítulo 4 é apresentada a implementação em CUDA e OpenMP, com as melhorias propostas e os resultados obtidos.

CAPÍTULO
2

Arquiteturas paralelas

Na primeira década do século 21, a computação sequencial encontrou uma barreira que impede o crescimento exponencial da quantidade de cálculos realizados. Esta é a barreira é o limite físico do tamanho dos transistores. Uma alternativa para continuar expandindo o poder de processamento é por meio do paralelismo computacional. A computação paralela é o modo de realizar cálculos simultâneos, dividindo problemas grandes em subproblemas menores que são resolvidos paralelamente. Computador paralelo é o conjunto de processadores organizados sob um determinado modelo de interconexão que os permitam coordenar atividades e trocas de informações. Os modelos de computação paralela são classificados de acordo com o nível de paralelismo de instruções e de dados. Neste capítulo são mostrados alguns desses modelos e bibliotecas para computação paralela.

2.1 Modelo PRAM

O modelo PRAM (Parallel Random-Access Machine) [20] é uma extensão natural do modelo de computação sequencial de von Neumann. Neste modelo temos uma grande quantidade de processadores, cada um com sua memória local, executando um programa local e toda a comunicação é feita através de troca de dados por uma unidade de memória global compartilhada. Cada processador tem um identificador único, acessível localmente. No modelo PRAM todos os processadores operam de modo síncrono sob o controle de um *clock* comum.

Um algoritmo desenvolvido para o modelo PRAM abstrato pode ser do tipo *single instruction multiple data* (SIMD) ou *multiple instruction multiple data*

(MIMD). Na variante SIMD, todos os processadores executam o mesmo programa a cada unidade de tempo, mas sobre dados diferentes. Além disso, o modelo permite que diferentes programas possam ser carregados nas memórias locais do processador, contanto que os processadores possam operar sincronamente.

Existem três variações do modelo PRAM baseados na forma de permissão de leitura e escrita na memória compartilhada. O *exclusive read exclusive write* (EREW) PRAM que não permite acesso simultâneo a um único local da memória. O *concurrent read exclusive write* (CREW) PRAM que permite acesso simultâneo apenas à leitura de instruções. O *concurrent read concurrent write* (CRCW) PRAM permite acesso simultâneo à leitura e escrita.

2.2 Memória Distribuída com MPI

Além do modelo de memória compartilhada, há também o modelo de memória distribuída, onde cada processador tem sua própria memória local e compartilha seus dados via troca de mensagens. Neste modelo, uma coleção de p processadores, cada um com sua própria memória local, se comunica através de uma rede interconectada. Aqui, a latência da rede interconectada pode ser menos crítica, já que cada processador provavelmente acessa sua própria memória local a maioria do tempo. Entretanto, a largura da banda de comunicação da rede pode ou não ser crítica, dependendo do tipo da aplicação paralela e a extensão da interdependência das tarefas.

O MPI (*Message Passing Interface*) é uma especificação para bibliotecas de troca de mensagem que implementa a comunicação entre os processadores de um *cluster*. Foi desenvolvido por um fórum internacional aberto consistindo de representantes da indústria, acadêmicos e laboratórios de governos. Foi rapidamente aceito por ter sido cuidadosamente especificado para permitir máximo desempenho em uma grande variedade de sistemas. As especificações foram definidas para programas em C/C++, Fortran, Java, entre outras. O MPI tem como vantagem a portabilidade, praticidade, eficiência e flexibilidade.

As aplicações que utilizam a biblioteca do MPI, precisam iniciá-lo através da função *MPI_Init* e encerrá-lo com *MPI_Finalize*. O MPI ao ser inicializado cria um canal comunicador entre os processadores, identificando-os no processo. As funções utilizadas para isso são *MPI_Comm_rank* e *MPI_Comm_size*. A comunicação entre os processadores através do MPI é feita por chamada de funções que "envelopam" as informações que precisam ser enviadas a outros processadores que por sua vez "desenvelopam" os dados e os utilizam. As funções de envio de mensagem podem ser bloqueantes ou não bloqueantes. As funções de envio de mensagem são *MPI_Send* e *MPI_Isend*. A primeira é o

envio bloqueante e a segunda é o não bloqueante. As funções de recebimento de mensagem são *MPI_Recv* e *MPI_Irecv*. Também o primeiro é o recebimento bloqueante e o segundo é o não bloqueante. Esses são os principais comandos dentre as centenas de funções que a biblioteca tem implementada.

2.3 Memória compartilhada com OpenMP

OpenMP é uma *application programming interface* (API) para máquinas *multicore* cujas funcionalidades são baseadas em esforços anteriores para facilitar a programação paralela em memória compartilhada [4]. Em vez de um padrão oficial sancionado, é um acordo alcançado entre os membros da *Architecture Review Board* (ARB), que compartilham interesse em uma abordagem portável, amigável e eficiente para a programação paralela em memória compartilhada. O OpenMP visa ser apropriado para implementação em uma ampla variedade de arquiteturas de *Symmetric Multiprocessing* (SMP).

Como seus predecessores, OpenMP não é uma nova linguagem de programação. Em vez disso, é uma notação para diretivas que pode ser adicionada a um programa sequencial em Fortran, C, ou C++. Essas notações tem o objetivo de descrever como o trabalho deve ser compartilhado entre *threads*. As *threads* executarão em diferentes processadores ou núcleos e ordenam o acesso à memória compartilhada conforme necessário. A inserção apropriada das funcionalidades do OpenMP em um programa sequencial permite que muitas aplicações se beneficiem das arquiteturas paralelas de memória compartilhada, em geral, com poucas modificações no código. Na prática, muitas aplicações têm um paralelismo considerável que pode ser explorado.

O sucesso do OpenMP pode ser atribuído a um número de fatores. Um é sua forte ênfase na programação paralela estruturada. Outro é que o OpenMP é comparativamente simples de usar, uma vez que o árduo trabalho de detalhar o programa em um código paralelo fica com o compilador. Tem como maior vantagem ser amplamente adotado, de modo que uma aplicação OpenMP executará em muitas plataformas diferentes.

A API do OpenMP foi desenvolvida para permitir a programação paralela em memória compartilhada. Ela suporta a paralelização de aplicativos variados. Além disso, seus criadores pretendiam fornecer uma abordagem que era relativamente fácil para aprender assim como aplicar. A API é projetada para permitir a abordagem incremental para paralelizar um código existente, na qual porções de um programa são paralelizados, possibilitando passos sucessivos. Este é um contraste marcante com a abordagem da conversão de um programa completo em um único passo que é tipicamente exigido pelos outros paradigmas de programação paralela. Ela foi considerada altamente

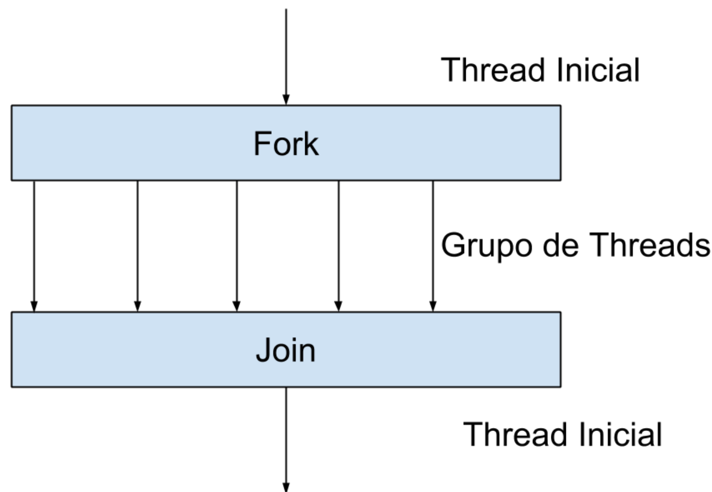


Figura 2.1: O modelo de programação *fork-join* suportado pelo OpenMP.

desejável para permitir que programadores trabalhem um único código fonte, ou seja, se um único conjunto de códigos fonte contém tanto o código para a versão sequencial quanto para a paralela do programa, então a manutenção é grandemente simplificada. Ao seguir estes objetivos, a API do OpenMP foi conduzida até o formato atual, e continua a guiar o desenvolvimento de novas funcionalidades.

Uma *thread* é uma entidade de tempo de execução que é capaz de executar independentemente um fluxo de instruções. O OpenMP aproveita uma grande quantidade de trabalho paralelizável, para especificar programas que serão executados por uma coleção de *threads* colaborativas. O sistema operacional cria um processo para executar um programa, ou seja, ele alocará alguns recursos para aquele processo, incluindo páginas de memória e registradores para manter os valores dos objetos. Se múltiplas *threads* colaboram para a execução de um programa, elas irão compartilhar os recursos, incluindo o espaço de endereçamento, do processo correspondente. As *threads* individuais precisam apenas de alguns recursos para si mesmas, para um contador e uma área na memória para salvar variáveis que são específicas para ela, includindo registradores e uma pilha. Múltiplas *threads* podem ser executadas em um único processador ou *core* via mudança de contextos. Além disso elas podem ser intercaladas via *multithreading* simultâneo. As *threads* executando simultaneamente em múltiplos processadores ou *cores* podem trabalhar concorrentemente para executar um programa paralelo.

Programas *multithread* podem ser escritos de várias maneiras, algumas das quais permitem interações complexas entre *threads*. O OpenMP tenta facilitar a programação e ajudar o usuário a evitar uma variedade de potenciais erros

de programação oferecendo uma abordagem estruturada para a programação *multithread*. Ele suporta o chamado modelo de programação *fork-join*, que é ilustrado na Figura 2.1. Sob esta abordagem, o programa inicia como uma única *thread* de execução, assim como um programa sequencial. A *thread* que executa este código é referida como a *thread inicial*. Sempre que uma construção paralela OpenMP é encontrada por uma *thread* que está executando o programa, ela cria um grupo de *threads* (este é o *fork*), torna-se o mestre do grupo, e colabora com os outros membros do grupo para executar o código dinamicamente cercado pelo construtor. No fim do construtor, apenas a *thread* original, ou mestre do grupo, continua; todas as outras terminam (este é o *join*). Cada porção do código cercado por uma construção paralela é chamada de região paralela.

O OpenMP espera que o desenvolvedor da aplicação dê uma especificação de alto nível para o paralelismo no programa e o método de explorar esse paralelismo. Assim, ele fornece uma notação para indicar as regiões de um programa OpenMP que devem ser executadas em paralelo, além de possibilitar o fornecimento de informações adicionais sobre como isto deve ser realizado. O trabalho de uma implementação OpenMP é resolver os detalhes de baixo nível para realmente criar *threads* independentes que executam o código e atribuir trabalho para elas de acordo com a estratégia especificada pelo programador.

Entre as funcionalidades que o OpenMP fornece estão:

- Criar grupos de *threads* para execução paralela;
- Especificar como compartilhar trabalho entre os membros do grupo;
- Declarar variáveis compartilhadas e privadas;
- Sincronizar *threads* e permitir que elas realizem certas operações independentemente.

A API do OpenMP inclui um conjunto de diretivas de compilador, biblioteca de rotinas de tempo de execução e variáveis de ambiente para especificar o paralelismo de memória compartilhada em programas Fortran e C/C++. Uma diretiva OpenMP é um comentário ou *pragma* especialmente formatado, que geralmente se aplica ao código subsequente no programa. Uma diretiva ou rotina OpenMP geralmente afeta apenas aquelas *threads* que a encontram. Muitas diretivas são aplicadas ao *bloco estruturado* do código, uma sequência de declarações executáveis com uma única entrada no topo e uma única saída ao fim, em programas Fortran, e uma declaração executável em C/C++. Em outras palavras, o programa não pode ramificar ou bifurcar blocos de código associados com as diretivas. Em programas Fortran, o início e fim do bloco de código são marcados explicitamente pelas diretivas OpenMP. Uma vez que o fim do bloco é explícito em C/C++, apenas o início precisa ser marcado.

2.4 CUDA

CUDA (*Compute Unified Device Architecture*) é uma arquitetura de computação paralela para Grafic Processor Units (GPUs) Nvidia. No passado, a GPU foi usada apenas como um co-processador da CPU para responder os muitos trabalhos gráficas em tempo real. Agora graças ao aumento de poder computacional das GPUs, elas também são muito eficientes em vários trabalhos de dados paralelos.

O SDK CUDA é uma extensão para a linguagem C que possibilita que código da GPU seja escrito em C. O código tem como alvo tanto o processador do *host* (a CPU) quanto o processador do dispositivo (a GPU). O processador do *host* gera tarefas *multithread*, ou *kernels* como são conhecidos em CUDA, para o dispositivo GPU. A GPU tem seu próprio gerenciador que irá então alocar os *kernels* para qualquer hardware GPU que esteja disponível.

Essa arquitetura é ideal para um problema fortemente paralelo, onde pouco eu nenhuma comunicação *interthread* ou *interblock* é necessária. Ela suporta comunicação *interthread* com primitivas explícitas usando recursos *onchip*. A comunicação *interblock*, no entanto, é suportada apenas pela invocação de vários *kernels* em série, e a comunicação entre os *kernels* é executada com memória global *off-chip*. Também pode ser realizada de uma maneira um pouco restrita por meio de operações atômicas de entrada e saída na memória global.

Os problemas são divididos em *grids* de blocos, cada um contendo múltiplas *threads*. Os blocos podem executar em qualquer ordem. Apenas um subconjunto dos blocos será executado em cada momento. Um bloco precisa ser executado do início ao fim em cada SM (multiprocessador simétrico). Os blocos são alocados da *grid* de blocos para qualquer SM que tiver espaços livres. Inicialmente isto é feito no formato *round-robin* de modo que cada SM obtém uma igualdade de distribuição de blocos. Para a maioria dos *kernels*, o número de blocos necessários será na ordem de oito ou mais vezes o número de SMs físicos na GPU. Na Figura 2.2 [6] vê-se um exemplo de processador simétrico e de outros blocos chave, como memória (global, textura, constante e compartilhada) e SP (*streaming processor*).

Utilizando uma analogia militar, temos um exército (uma *grid*) de soldados (*threads*). O exército é dividido em um número de unidades (*blocks*), cada um comandado por seu tenente. A unidade é dividida em pelotões de 32 soldados (um *warp*), cada um comandado por um sargento.

Para executar alguma ação, o comando central (o programa *kernel* ou *host*) deve fornecer alguma ação e alguns dados. Cada soldado (*thread*) trabalha em sua parte individual do problema. *Threads* podem de tempos em tempos

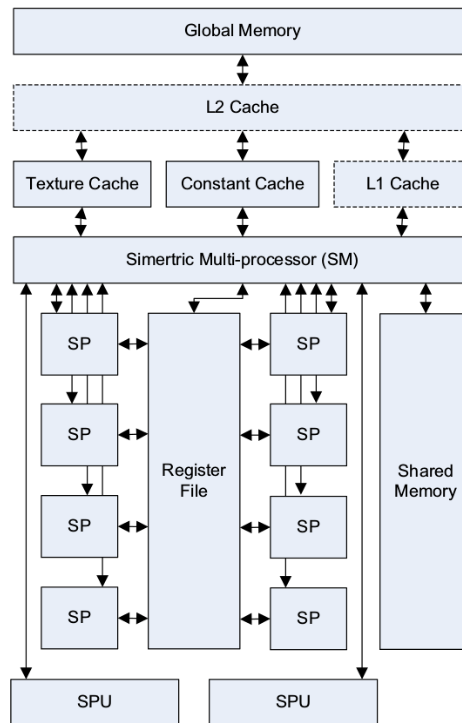


Figura 2.2: Por dentro de um SM.

trocar dados umas com as outras sob a coordenação tanto do seu sargento (o *warp*) ou do seu tenente (o bloco). Entretanto, qualquer coordenação com outras unidades (blocos) deve ser executada pelo comando central (o programa *kernel* ou *host*).

Assim, é necessário pensar na orquestração de milhares de *threads* nesta maneira hierárquica quando planejar a implementação de um programa CUDA. Isso pode parecer um pouco complexo, mas para a maioria dos programas fortemente paralelos é apenas um caso de pensar que cada *thread* gera um único dado de saída. Uma GPU típica tem cerca de 24 mil *threads* ativas. Numa GPU Fermi é possível definir 65.535 x 65.535 x 1536 *threads* no total, das quais 24 mil estarão ativas em qualquer momento. Isso geralmente é suficiente para cobrir a maioria dos problemas em um único nó.

A CUDA C permite definir funções em C chamadas *kernels* que podem ser executadas em paralelo. O programa C é executado no *host* (ou CPU) pela *thread* do *host*. Os *kernels* (programas CUDA) são iniciados pela *thread* do *host* e executam no *device* (ou GPU) pelas muitas *threads* CUDA. O número de *threads* executando um *kernel* é definido pelo programador. As *threads* em execução são divididas em blocos tridimensionais. O conjunto de blocos forma uma *grid* tridimensional.

Um exemplo da declaração de um *kernel* é:

```
__global__ void kernel_function(int* data);
```

que precisa ser chamado dessa forma:

```
kernel_function<<<grid_size,block_size>>>(algumArray);
```

Para cada *thread* o índice único pode se calculado na *grid* da seguinte maneira:

```
int this_id = (threadsInBlock * nBlockInGrid) + nThreadInBlock;
```

O host e o device possuem espaços de memória separados chamados *host memory* e o *device memory*, ambos residindo na *dynamic random-access memory* (DRAM). Existem vários tipos de memória compartilhada: *global*, *local*, *compartilhada*, *constante* e *textura*. Há também vários *registradores*. A memória compartilhada e os registradores estão localizadas no chip da GPU. As outras estão localizadas fora do *chip* de modo que possuem alta latência de acesso. Entretanto, os tamanhos da memória compartilhada e dos registradores são muito menores do que das outras. Uma vez que a memória local é localizada fora do chip, então sua latência de acesso também é grande.

Os escopos e os tempos de vida da memória local e os registradores são restritos a uma *thread*. O escopo da memória compartilhada e seu tempo de vida é restrito a todas as *threads* do bloco. O tempo de vida e o acesso a outras memórias estão disponíveis para todos as *threads* iniciadas e para o *host*.

As *threads* podem ser sincronizadas no escopo de um bloco pela função *__syncthreads()*. Ela define um semáforo que faz com que a execução do código adicional espere até que todas as *threads* paralelas alcancem o ponto específico.

Existem duas funções atômicas que podem ser utilizadas: *atomicAdd()* e *atomicSub()*, que realizam as operações *read-modify-write* em *words* 64-bit residindo na memória global. As operações atômicas são mais lentas do que as não-atômicas mas permitem a implementação de programas sem qualquer sincronização entre as *threads*.

A largura de banda é a taxa na qual os dados podem ser transferidos. A largura de banda entre a memória global no dispositivo e a memória global no *host* é muito menor do que a largura de banda entre a memória global no dispositivo e o espaço de memória na GPU. Portanto, é importante minimizar a transferência de dados entre o dispositivo e o *host*. Por isso, é importante realizar a cópia apenas dos dados importantes para a GPU.

Para alocar e desalocar memória no dispositivo existem as funções *cudaMalloc()* e *cudaFree()* e para copiar memória entre o *device* e o *host* existe a função *cudaMemcpy()*.

CAPÍTULO
3

Algoritmos paralelos para o problema do Fluxo Máximo em Redes

Vários algoritmos paralelos foram propostos para o problema do fluxo máximo em redes. O algoritmo de Anderson e Setubal [1] utiliza um modelo de paralelismo SMP de memória compartilhada. Esse algoritmo apresenta uma heurística de rotulação global que possibilita bons resultados práticos. Este algoritmo é baseado no algoritmo sequencial de *push-relabel*. Neste capítulo são apresentados alguns algoritmos para o fluxo máximo utilizando arquiteturas paralelas.

3.1 Algoritmos baseados em Multi-threaded

Hong [15] apresentou uma algoritmo *multi-threaded* em CPU livre de *locks* para o problema do fluxo máximo baseado na versão de Goldberg [12] do algoritmo *push-relabel*. A implementação do algoritmo de Hong exige uma arquitetura *multi-threaded* que suporta operações *read-modify-write* atômicas.

Sem perda de generalidade assume-se que o número de *threads* é $|V|$, que é o número de vértices, e que cada uma delas manipula exatamente um vértices do grafo, incluindo todas as operações *push* e *relabel* nele. Em geral, alguns vértices podem ser manipulados por uma única *thread*.

Seja u uma *thread* em execução representando o vértice $u \in V$. No algoritmo de Hong, cada uma das *threads* em execução tem os seguintes atributos privados. A variável e' armazena o excesso, ou total do fluxo que entra menos o total do fluxo que sai do vértice u. A variável h' armazena a altura do vértice

adjacente v de u tal que $(u,v) \in E_f$. A variável h'' armazena a altura do vizinho mais baixo v'' de u.

Outras variáveis são compartilhadas entre todas as *threads* em execução. Entre elas estão os arrays com excessos e alturas dos vértices e a capacidade residual das arestas.

Primeiro, a operação *Init* é realizada pela *thread* principal. Esse código de inicialização é o mesmo da versão sequencial do *push-relabel* correspondente. Em seguida, a *thread* principal inicia as *threads* para executar o algoritmo *push-relabel* paralelo livre de *lock*. A mudança básica, introduzida por Hong, lida com a seleção da operação (*push* ou *relabel*) que deve ser executada por u, e para qual dos vértices adjacentes $v'', (u,v'') \in E_f$ o fluxo deve ser empurrado. Ao contrário da operação *push* da versão sequencial genérica onde qualquer vértice v conectado por uma aresta residual para u tal que $h(u) = h(v) + 1$ pode ser empurrado, ele seleciona o vértice mais baixo entre todos os vértices conectados por arestas residuais. Em seguida, se a altura de v'' for menor do que a altura de u, a operação *push* é realizada. De outro modo, a operação *relabel* é realizada, isto é, a altura de u é modificada para $h(v'') + 1$. Note que a operação *relabel* não precisa ser atômica porque apenas a *thread* u pode mudar o valor da altura de u. Além disso, todas as linhas críticas no código onde mais do que duas *threads* executam a instrução de escrita são atômicas. Consequentemente é fácil ver que o algoritmo está correto com respeito às instruções de leitura e escrita. O Algoritmo 1 descreve essas operações.

Todas as *threads* em execução têm acesso a uma variável crítica na memória global, mas na verdade suas tarefas são executadas sequencialmente, graças ao acesso atômico aos dados. A ordem das operações nesta sequência não pode ser prevista.

O algoritmo livre de *lock* termina após no máximo $O(V^2E)$ operações *push* e *relabel*. Por causa da execução em paralelo por muitas *threads*, a complexidade do algoritmo é analisada pelo número de operações e não o tempo de execução.

Um outro algoritmo *multi-threaded* foi apresentado por Soner e Ozturan [24]. Este algoritmo é baseado no Pmaxflow, que é um solucionador paralelo de fluxo máximo (https://code.google.com/p/pmaxflow) *open source* e paraleliza o algoritmo de Goldberg. Infelizmente, a primeira versão do Pmaxflow era vulnerável a problemas súbitos relacionados à concorrência. Na versão de Soner foram feitas modificações para superar esses problemas.

Durante o processamento dos vértices ativos (com fluxo excedente), a implementação Pmaxflow utiliza uma estratégia de seleção FIFO (First In First Out). Cada *thread* tem sua própria fila local e um mecanismo de transferência de tarefa é utilizado para balancear a carga das *threads*. Pmaxflow é imple-

Algoritmo 1 Algoritmo *push-relabel multi-threaded* livre de *lock* de Hong

1: **function** INIT()
2: $\quad h(s) \leftarrow |V|$
3: $\quad$ **for all** $u \in V - s$ **do**
4: $\qquad h(u) \leftarrow 0$
5: $\qquad e(u) \leftarrow 0$
6: $\quad$ **end for**
7: $\quad$ **for all** $(u,v) \in E$ **do**
8: $\qquad c_f(u,v) \leftarrow c(u,v)$
9: $\qquad c_f(v,u) \leftarrow c(v,u)$
10: $\quad$ **end for**
11: $\quad$ **for all** $(s,u) \in E$ **do**
12: $\qquad c_f(s,u) \leftarrow 0$
13: $\qquad c_f(u,s) \leftarrow c(u,s) + c(s,u)$
14: $\qquad e(u) \leftarrow c(s,u)$
15: $\quad$ **end for**
16: **end function**
17:
18: **function** LOCK-FREE *push-relabel*()
19: $\quad$ **while** $(e(u) > 0)$ **do**
20: $\qquad e' \leftarrow e(u)$
21: $\qquad v'' \leftarrow null$
22: $\qquad h'' \leftarrow \infty$
23: $\qquad$ **for all** $(u,v) \in E_f$ **do**
24: $\qquad\quad h' \leftarrow h(v)$
25: $\qquad\quad$ **if** $h' < h''$) **then**
26: $\qquad\qquad v'' \leftarrow v$
27: $\qquad\qquad h'' \leftarrow h'$
28: $\qquad\quad$ **end if**
29: $\qquad$ **end for** $\triangleright$ v'' é o vizinho mais baixo de u em E_f
30: $\qquad$ **if** $h(u) > h''$ **then** $\triangleright$ a *thread* u realiza PUSH em direção a v
31: $\qquad\quad d \leftarrow \min(e', c_f(u,v''))$
32: $\qquad\quad c_f(u,v'') \leftarrow c_f(u,v'') - d$
33: $\qquad\quad c_f(v'',u) \leftarrow c_f(v'',u) + d$
34: $\qquad\quad e(u) \leftarrow e(u) - d$
35: $\qquad\quad e(v'') \leftarrow e(v'') + d$
36: $\qquad$ **else** $\triangleright$ a *thread* u realiza RELABEL
37: $\qquad\quad h(u) \leftarrow h'' + 1$
38: $\qquad$ **end if**
39: $\quad$ **end while**
40: **end function**

mentado utilizando Pthreads, mas também faz uso de construções *Threading Building Blocks* (TBB) para variáveis atômicas. A principal rotina do Pmaxflow implementa basicamente um laço *while* que é repetido até que não permaneçam vértices ativos nas filas locais das *threads*. No corpo do principal laço *while* uma ou mais das seguintes operações podem ser executadas:

- *Rotulação global paralela*: isto é executado periodicamente em toda a rede por todas as *threads* para calcular os rótulos de distância de cada vértice para o sorvedouro *t*. Esta operação é basicamente uma busca em largura paralela na rede partindo do sorvedouro. Todas as *threads* sincronizam por uma barreira antes e depois da operação de rotulação global. A rotulação global é normalmente executada após *n* operações de rotulação. Entretanto, ela também pode ser acionada por uma *thread* se a propriedade do rótulo de distância é violada. Isto pode ocorrer como uma resultado das execuções intercaladas concorrentes das operações *push/relabel*;

- *Transferência de tarefa*: uma *thread* pode esperar até que uma tarefa seja passada para ela por outras *threads*. Isto é feito de modo a balancear as cargas das *threads*;

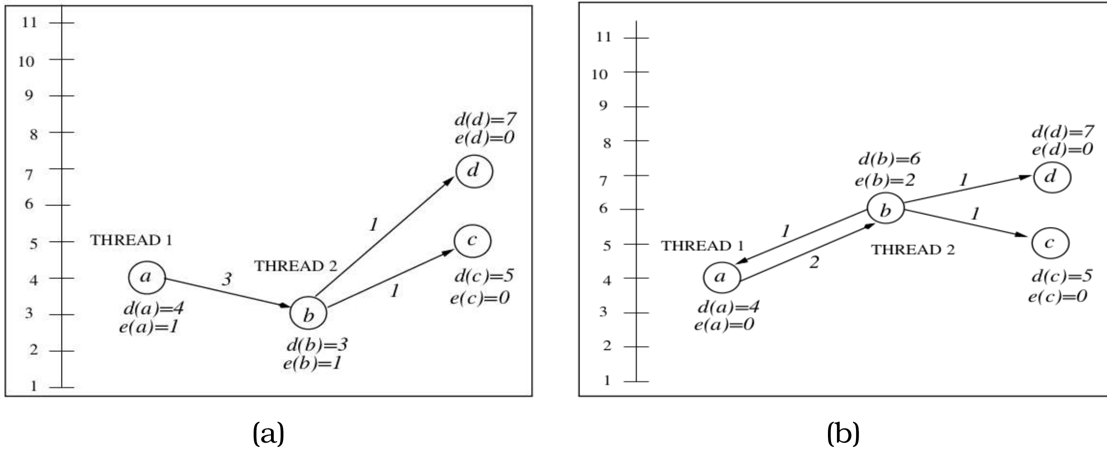
- *Operação* push: empurra fluxo sobre uma aresta *(u,v)* de um vértice com excesso *u* para seu vértice vizinho *v*, que possui o menor rótulo de distância, ou seja, o vizinho adjacente que está mais próximo do sorvedouro;

- *Operação* relabel: aumenta o rótulo de distância de um vértice ativo *p* para ser uma unidade a mais do que a distância dos vértices adjacentes, ou seja:

$$h(u) = min\{h(v)+1 : (u,v) \text{ é uma aresta de saída de } u\}.$$

Aqui, concentramos nas operações intercaladas concorrentes de *push* e *relabel* que podem levar a uma violação das propriedades de rótulo de distância. Este problema foi primeiramente observado por Hong [14]. Considere o exemplo na Figura 3.1 que ilustra como rótulos de distância inválidos podem surgir quando uma *thread* executa operações *push* no vértice *a* enquanto outra executa concorrentemente uma operação de rotulação no vértice *b*. Aqui, as notações $e(a)$ e $h(a)$ indicam fluxo excedente e rótulo de distância no vértice *a* respectivamente. No algoritmo de Goldberg, os rótulos dos vértices devem satisfazer à seguinte invariante em toda parte da execução do algoritmo: dado uma aresta (u,v), então deve haver $h(u) \leq h(v)+1$.

Rotulação inválida pode potencialmente acontecer devido aos seguintes fatores: enquanto a *thread* 2 que rotula o vértice b vê apenas os vértices c e



(a) (b)

Figura 3.1: Formação dos rótulos de distância inválidos: Na figura (a), o fluxo é empurrado em (a,b) enquanto b está sendo rotulado; e na figura (b) a aresta reversa (b,a) com $h(b) > h(a)+1$ é formada.

d como seus únicos vizinhos, a *thread* 1 pode empurrar fluxo para fora de a pela aresta (a,b) ao mesmo tempo e assim cria uma aresta residual reverso (b,a) sem que a *thread* rotulante 2 tenha ciência. O indício que leva a isso é o seguinte:

1. *Thread* 1: na linha 11 no método *push*, $arc.w.d$ é acessado atomicamente para testar a condição $arc.w.d = v.d - 1$ que é avaliada como verdadeira;

2. *Thread* 2: na linha 1 no método *relabel*, $v.d$ (ou seja, $w.d$ na visão da *thread* 1) é inicializado atomicamente para o valor máximo $2N$;

3. *Thread* 2: $arc.flow$ é acessado atomicamente na linha 4 do método *relabel* e a condição $arc.cap - arc.flow > 0$ é avaliada como verdadeira ou falsa (dependendo do valor de distância). Isto é feito duas vezes, uma para o vértice c e outra para o vértice d;

4. *Thread* 1: $arc.other.flow$ é atualizado atomicamente no $push$;

5. *Thread* 2: o rótulo de distância $v.d$ (ou seja, $w.d$ na visão da *thread* 1) é atualizado atomicamente no $relabel$.

O código do Pmaxflow era vulnerável neste problema sutil. A implementação do Pmaxflow foi remodelada por Soner e Ozturan para corrigir este problema de concorrência. Note que o Pmaxflow paraleliza o algoritmo paralelo original de Goldberg enquanto que Hong contribui com um algoritmo modificado. A modificação de Hong envolve mudar a condição sob a qual a operação $push$ é executada. O algoritmo original de Goldberg empurra fluxo de um vértice excedente para um vértice adjacente cujo rótulo de distância seja uma unidade menor. No algoritmo de Hong, entretanto, uma operação $push$ de um vértice excedente u pode ser feita para um vértice adjacente v cuja

distância $h(v)$ seja menor em uma ou mais unidades do que de $h(u)$, ou seja, $h(u) \geq h(v)+1$. Esta modificação é feita de modo a corrigir a saída da problemática operação concorrente de rotulação que causa a violação de uma propriedade invariante obedecida pelos rótulos de distância. O Pmaxflow ainda empurra fluxo para um vértice v com distância $h(v)$ menor em exatamente uma unidade, ou seja, $h(u) = h(v)+1$, como no algoritmo original de Goldberg e resolve a saída das problemáticas operações de rotulação concorrente iniciando um processo de rotulação global. Uma vez que a rotulação global recalcula as distâncias novamente, o problema das distâncias inválidas está resolvido. Note que, o fato do que entre os eventos (1) e (4) nos passos acima, a *thread* 1 executa apenas umas poucas instruções, isso significa que os eventos (2) e (3) da thread 2 devem ser espremidos entre estes eventos sucessivos muito próximos. Esta é uma situação em que a probabilidade de ocorrer pode ser muito baixa, embora possível. Portanto, se esta situação de fato ocorrer, a correção por um processo custoso como a rotulação global pode ser justificada.

Por isso, esta solução para o problema basicamente permite que a situação de distâncias inválidas ocorra, mas introduz declarações para detectar isto posteriormente assim como acionar a rotulação global. Após a rotulação de um vértice excedente, a operação *push* pode ser aplicada. As declarações nas linhas 1-8 que foram adicionadas na operação *push* avançam pela lista de vizinhos dos vértices incorretamente rotulados e verificam a condição de distância. Se for violada, então a *labelingcounter* é definida para um número maior que posteriormente aciona a rotulação global. Note que não é empurrado fluxo a partir deste vértice neste momento. Após a rotulação global, o fluxo pode ser empurrado deste vértice como de costume.

Enquanto esta solução possa parecer ad-hoc, ela tem a vantagem de permitir a paralelização do algoritmo original de Goldberg. O algoritmo de Hong encontra o fluxo máximo com $O(V^2E)$ operações de *push* e *relabel* enquanto que o algoritmo baseado em FIFO de Goldberg paralelo tem o número menor de $O(V^3)$ operações. Note que E é o número de arestas que no pior caso pode ser $O(V^2)$. Portanto, esta implementação pode oferecer vantagens especialmente quando o número de arestas é muito grande.

3.2 Algoritmos baseados em CUDA

3.2.1 Push-relabel

Várias implementações do fluxo máximo baseadas em GPU foram propostas. O trabalho de Hussein, Varshney e Davis [17], além do de Vineet e Narayanan [25] apresentaram soluções restritas do fluxo máximo aplicáveis apenas a grids.

Hong e He [16] desenvolveram uma implementação para grafos genéricos. Eles melhoraram o algoritmo *push-relabel* livre de *lock*, utilizando CUDA e adicionando a heurística da rotulação global sequencial realizada na CPU (Algoritmo 2). De acordo com os autores e os resultados de seus experimentos, o novo esquema CPU-GPU-Híbrido do algoritmo *push-relabel* livre de *lock* é robusto e eficiente.

Algoritmo 2 Funções *init* e *push-relabel-cpu* do algoritmo CPU-GPU-Híbrido

```
 1: function INIT()
 2:     inicie e, h, c_f e ExcessTotal
 3:     copie e e c_f da memória principal da CPU para a memória global CUDA
 4: end function
 5:
 6: function PUSH-RELABEL-CPU()
 7:     while (e(s) + e(t) < ExcessTotal) do
 8:         copie h da memória principal da CPU para a memória global CUDA
 9:         execute push-relabel-kernel()
10:         copie e, c_f e h da memória global CUDA para a memória principal da
    CPU
11:         execute global-relabel-cpu()
12:     end while
13: end function
```

De modo similar ao Algoritmo 1, pode-se assumir que cada vértice é operado por no máximo uma *thread*. A versão anterior do algoritmo *push-relabel* livre de *lock* será chamado de algoritmo genérico, enquanto o esquema CPU-GPU-Híbrido será chamado algoritmo híbrido.

A inicialização do algoritmo híbrido é a mesma de seu correspondente no algoritmo genérico. O algoritmo híbrido mantém 3 arrays com excessos, alturas e capacidades residuais dos vértices e arestas e a variável global $ExcessTotal$, que é igual ao valor do fluxo empurrado a partir da fonte. $ExcessTotal$ reside na memória global do host e pode ser alterada durante a rotulação global.

Ao contrário do algoritmo genérico, o corpo principal do algoritmo híbrido é controlado pela *thread* host na CPU. A *thread* host executa o laço *while* até que o valor acumulado do excesso armazenado na fonte e no sorvedouro alcancem o valor de $ExcessTotal$. Neste momento todo o fluxo válido chega ao sorvedouro e o resto do fluxo retorna à fonte. Então, o excesso no sorvedouro é igual ao valor do fluxo máximo.

No primeiro passo do laço *while*, a *thread host* copia as alturas dos vértices para o *device* e inicia a função *push-relabel-kernel*. Quando o controle é devolvido para a *thread host*, o pseudo-fluxo calculado e as alturas dos vértices são copiados para a memória da CPU e é realizada a função *global-relabel-cpu*.

A função *push-relabel-kernel*, conforme o Algoritmo 5, difere do algoritmo genérico no tempo de execução do *kernel*. A *thread* encerra a execução do

Algoritmo 3 Algoritmo *push-relabel* CPU-GPU-Híbrido dinamicamente ajustado

1: Inicializa e, h, c_f, $ExcessTotal$, $aSize$ e $Threshold$
2: $UltimoDispositivo \leftarrow CPU$
3: **while** $e(s) + e(t) < ExcessTotal$ **do**
4: **if** $aSize > Threshold$ **then**
5: **if** $UltimoDispositivo = CPU$ **then**
6: copie e e c_f da memória principal da CPU para a memória global CUDA
7: $UltimoDispositivo \leftarrow GPU$
8: **end if**
9: copie h da memória principal da CPU para a memória global CUDA
10: *push-relabel-kernel()*
11: copie c_f, h e e da memória global CUDA para a memória principal da CPU
12: *global-relabel-cpu()*
13: **else**
14: **if** $UltimoDispositivo = GPU$ **then**
15: $UltimoDispositivo \leftarrow CPU$
16: **end if**
17: Versão serial de Goldberg incluindo operações de rotulação global e gap **hipr**
18: **end if**
19: $Threshold \leftarrow \frac{T_{overhead} C_{cuda} C_{cpu}}{C_{cuda} - C_{cpu}}$
20: **end while**

Algoritmo 4 Inicialização do algoritmo CPU-GPU-Híbrido

1: **function** INIT()
2: $h(s) \leftarrow |V|$
3: $e(s) \leftarrow 0$
4: **for all** $u \in V - \{s\}$ **do**
5: $h(u) \leftarrow 0$
6: $e(u) \leftarrow 0$
7: **end for**
8: **for all** $(u,v) \in E$ **do**
9: $c_f(u,v) \leftarrow c(u,v)$
10: $c_f(v,u) \leftarrow c(v,u)$
11: **end for**
12: **for all** $(s,u) \in E$ **do**
13: $c_f(s,u) \leftarrow c_f(s,u) - c(s,u)$
14: $c_f(u,s) \leftarrow c_f(u,s) + c(s,u)$
15: $e(u) \leftarrow c(s,u)$
16: $ExcessTotal \leftarrow ExcessTotal + c(s,u)$
17: **end for**
18: **end function**

laço *while* após *cycle* interações (onde *cycle* é uma constante inteira definida pelo usuário) e não quando o vértice se torna inativo. Após parar o laço, a rotulação global é executada e então o laço é inicializado novamente.

Uma vez que o laço pode terminar a qualquer momento (de modo aleatório se comparado à computação do fluxo sequencial) pode ocorrer que a propriedade de alguma aresta residual $(u,v) \in E_f$ seja violada, por exemplo $h(u) > h(v)+1$. Então, antes de calcular as novas alturas dos vértices, todas as arestas nessa situação precisam ser canceladas empurrando o fluxo. Isso é feito nas primeiras linhas da função *global relabeling*. Em seguida, atribuem-se novas alturas aos vértices executando uma busca em largura reversa a partir do sorvedouro em direção a fonte. As novas alturas são iguais as menores distâncias no grafo residual. O excessos dos vértices, que não estão disponíveis a partir do sorvedouro na árvore reversa de busca em largura, precisam ser subtraídos de $ExcessTotal$, porque armazenam um excesso que nunca alcançará o sorvedouro.

Algoritmo 5 Função push-relabel-kernel

```
 1: function PUSH-RELABEL-KERNEL()                    ▷ Nó u operado pela thread u
 2:     while (cycle > 0) do
 3:         if (e(u) > 0 and h(u) < |V|) then
 4:             e' ← e(u)
 5:             h' ← ∞
 6:             for all (u,v) ∈ E_f do
 7:                 h'' ← h(v)
 8:                 if h'' < h' then
 9:                     v' ← v
10:                     h' ← h''
11:                 end if
12:             end for
13:             if h(u) > h' then
14:                 delta ← min(e', c_f(u,v'))
15:                 AtomicAdd(c_f(v',u), delta)
16:                 AtomicSub(c_f(u,v'), delta)
17:                 AtomicAdd(e(v'), delta)
18:                 AtomicSub(e(u), delta)
19:             else
20:                 h(u) ← h' + 1
21:             end if
22:         end if
23:         cycle ← cycle − 1
24:     end while
25: end function
```

Hong e He melhoraram ambas as heurísticas para um novo método de rotulação global assíncrona (ARG) [14]. Até agora, as heurísticas de rotulação global e *gap* executavam independentes do algoritmo livre de *lock* (na imple-

mentação em CUDA, para executar as heurísticas o controle é retornado para a CPU). A principal razão para isso foi que as operações de *push* e *relabel* são mutualmente exclusivas com as heurísticas de rotulação global e gap. Entretanto, esse problema não ocorre nas versões não livres de *lock* do algoritmo *push-relabel*. Na nova abordagem de Hong e He, a heurística ARG é executada por uma *thread* distinta que corresponde a qualquer vértice e é executada periodicamente, enquanto as outras *threads* executam assincronamente operações *push* e *relabel*. Isso melhora significantemente o tempo de execução do algoritmo. O único problema é que a heurística ARG mantém uma fila dos vértices não visitados cujo tamanho é $O(V)$. Na programação em CUDA, uma fila desse tamanho pode ser mantida apenas na memória global cujo acesso é muito lento. Talvez seja por isso que a implementação apresentada em [14] utiliza C e a biblioteca *pthread* para as construções *multi-threaded*.

Algoritmo 6 Rotulação global para CPU-GPU-Híbrido

1: **function** GLOBAL-RELABELING()
2: **for all** $(u,v) \in E$ **do**
3: **if** $h(u) > h(v) + 1$ **then**
4: $e(u) \leftarrow e(v) - c_f(u,v)$
5: $e(u) \leftarrow e(v) + c_f(u,v)$
6: $c_f(v,u) \leftarrow c_f(v,u) + c_f(u,v)$
7: $c_f(u,v) \leftarrow 0$
8: **end if**
9: **end for**
10: Faça uma Busca em Largura reversa a partir do sorvedouro e atribua a função de altura com cada nível do vértice na árvore de busca.
11: **if** nem todos os vértices foram rotulados **then**
12: **for all** $u \in V$ **do**
13: **if** u não foi rotulado e marcado **then**
14: marque u
15: $ExcessTotal \leftarrow ExcessTotal - e(u)$
16: **end if**
17: **end for**
18: **end if**
19: **end function**

3.2.2 Autoestabilizante

Outro algoritmo baseado em CUDA para o problema do fluxo máximo foi desenvolvido por Silva [7]. Esta abordagem inspira-se no algoritmo autoestabilizante de Gosh [13] para produzir uma implementação usando CUDA. Neste algoritmo define-se que existe um processador para cada vértice de G. Cada aresta (u,v) pertencente a $E(G)$ corresponde a um link entre o processador u e v. Então temos que a rede de processadores terá a forma do grafo G. Cada

processador u tem um conjunto de variáveis que só podem ser escritas por u, mas podem ser lidas por qualquer processador vizinho de u, incluindo u. Para cada processador u temos:

- A demanda(u) é a soma de todo o fluxo que sai do vértice, menos o fluxo que entra no vértice; também temos que demanda$(t) = \infty$, sendo t o sorvedouro;

- Executa as mesmas instruções menos o processador s que fica inativo;

- Executa 4 ações que são ativadas de acordo com certas condições, estas dependem das suas variáveis locais ou de seus vizinhos;

- Temos que $h(u)$ é o tamanho do menor caminho a s até o momento no grafo residual.

Para cada aresta (u,v) temos $f(u,v)$ que é o fluxo que percorre a direção u para v, onde somente os processadores u e v podem ler e escrever.

Neste algoritmo temos as 4 funções, chamadas funções de guarda, que determinam as ações de cada processador:

- Função S1: cada vértice u calcula sua distância $h(u)$ examinando as distâncias dos vizinhos e escolhendo a menor distância $h(v)$; se $min(h(v)+1,V)$ for diferente de $h(u)$ então $h(u) \leftarrow min(h(v)+1,V)$. Isso porque um vértice só pode empurrar fluxo para outro que tenha distância menor em pelo menos uma unidade;

- Função S2: para qualquer vértice $u \neq s$, se demanda$(u) < 0$ então diminui-se o valor do fluxo que entra em u no valor absoluto da demanda(u) realizando um *Reduce_InFlow*;

- Função S3: para qualquer vértice u se demanda$(u) > 0$ e $h(u)$ menor que a quantidade de vértices em N então u tenta buscar fluxo através de uma aresta fornecedora (v,u) que aparenta ter a menor distância de s e então aumentamos o fluxo nessa aresta com o mínimo entre demanda(u) e $r(v,u)$, sendo $r(v,u)$ a capacidade residual da aresta (v,u);

- Função S4: para qualquer vértice $u \neq t$, se demanda$(u) > 0$ e $h(u)$ igual ao número de vértices no grafo, então acredita-se que não existe caminho entre s e u no grafo residual então a demanda excedente não pode ser atendida, logo é necessário diminuir o fluxo que sai do vértice, realizando um *Reduce_OutFlow*. Em resumo, temos as seguintes definições:

- $IN(u) = \{v|(v,u) \in A_f\}$, onde A_f são as arestas do grafo residual;

- $D(u) = \{h(p)+1|p \in IN(u)\}$, onde $IN(u)$ é o conjunto dos vértices que podem empurrar fluxo para u e $D(u)$ é a menor distância dentre estes vértices, acrescido em uma unidade;

- $pull(v,u) = (demanda(u) > 0) \wedge (h(u) < n) \wedge (h(v) = h(u) - 1)$;

- $push(u) = (demanda(u) > 0) \wedge (h(u) = n) \wedge (u \neq t)$;

Podemos resumir as quatro funções de guarda como:

- S1: se $h(u) \neq min(D(u) \cup V)$, então $h(u) \leftarrow min(D(u) \cup V)$;

- S2: se demanda(u) < 0, então realizamos um *Reduce_InFlow(u)*;

- S3: se $\exists_v \in IN(u,v) : pull(v,u)$, então $f(v,u) \leftarrow f(v,u) + min(demanda(u), r(v,u))$;

- S4: se $push(u)$, então realizaremos um *Reduce_OutFlow(u)*;

Algoritmo 7 Reduce_InFlow

1: **procedure** REDUCE_INFLOW(i)
2: Encontre $(k,u) \in A$ dado que $f(k,u) > 0$
3: $f(k,u) \leftarrow f(k,u) - min(-\texttt{demanda}(u), f(k,u))$
4: **end procedure**

Algoritmo 8 Reduce_OutFlow

1: **procedure** REDUCE_OUTFLOW(i)
2: Encontre $(u,k) \in E$ dado que $f(u,k) > 0$
3: $f(u,k) = f(u,k) - min(\texttt{demanda}(u), f(u,k))$
4: **end procedure**

O algoritmo autoestabilizante funciona da seguinte forma: primeiro todos as arestas do grafo são saturadas, ou seja, o valor do fluxo é o mesmo da capacidade total da aresta, em seguida cada processador executa as 4 funções de guarda com o objetivo de encontrar o fluxo máximo. S1 é a função responsável por estabelecer qual é a distância do vértice até a fonte no grafo residual. S2 tem a responsabilidade de devolver o fluxo que está sobrando, reduzindo o fluxo entrante no vértice com o valor absoluto da demanda. S3 busca fluxo através de uma aresta fornecedor quando existe a demanda a ser atendida. S4 quando supõe-se que não há mais caminho de s a u no grafo residual, a demanda excedente não pode ser atendida, sendo necessário diminuir o fluxo que sai do vértice. Cada processador executa as quatro funções enquanto houver pelo menos uma função que tenha suas condições atendidas, caso contrário o algoritmo para e obtemos o fluxo máximo.

Figura 3.2: Um exemplo de particionamento de rede.

3.3 Algoritmos baseados em PRAM

Caragea e Vishkin [3] desenvolveram uma arquitetura *multi-core* inspirada em PRAM, chamada *explicit multi-threading* (XMT). Essa arquitetura possibilitou a implementação do algoritmo *push-relabel* para o fluxo máximo baseado em PRAM de Shiloach e Vishkin [23]. Utilizando XMT, o algoritmo tornou-se mais fácil de programar e alcançou *speedups* maiores dos que os trabalhos anteriores, quando comparados com a melhor implementação sequencial.

Para a implementação do algoritmo, foi utilizada uma extensão de C, o XMTC, sigla para *explicit multi-threading C*. Essa extensão permite a programação orientada a PRAM. Essa implementação é completamente livre de *lock*, permitindo escalar para muito mais *cores*. As operações *push* e *relabel* são divididas em duas fases, uma para calcular e outra para propagar os resultados. A operação atômica *prefix-sum* é utilizada para coordenar o acúmulo de fluxo vindo de várias fontes. O volume de trabalho foi reduzido por manter uma lista de todos os vértices ativos e utilizar uma *thread* por vértice ativo. Uma busca em largura paralela é executada periodicamente no grafo residual para realizar a rotulação global.

3.4 Algoritmos baseados em MPI

No trabalho de Jincheng e Lixin [18] é apresentado um algoritmo paralelo com MPI para o problema do fluxo máximo, assumindo que um grafo com V vértices e E arestas é particionado em várias regiões, ou partições. O fluxo dentro de cada região é empurrado para sua fronteira, que é o limite entre uma região e outra, através da execução do algoritmo *hipr* iterativamente e é administrado um método especial para descarregar o fluxo para fora da região, reduzindo a passagem de mensagens através de três melhorias que são diferentes do algoritmo *push-relabel* comum. A Figura 3.2 mostra um exemplo

de particionamento da rede em três partes, onde as arestas pontilhadas são as arestas de fronteira.

Os passos do algoritmo paralelo MPI incluem: 1) não tentar calcular as distâncias entre os vértices de fronteira e o sorvedouro diretamente, mas por uma função de distância que calcula a distância entre a região e o sorvedouro, e o relacionamento entre o vértice de fronteira e suas regiões adjacentes; 2) o método para descarregar fluxo para fora da região varia conforme o fluxo nas arestas de fronteira, o estado dos vértices nas regiões adjacentes e a distância entre região e sorvedouro; 3) não tentar empurrar o fluxo pelo caminho que precisa de muita troca de mensagens. O algoritmo paralelo foi testado em vários tipos de redes utilizadas no primeiro Desafio de Implementação DIMACS (*Center for Discrete Mathematics and Theoretical Computer Science*) e descobriu-se que o algoritmo paralelo tem uma taxa de aceleração muito boa comparada ao algoritmo sequencial para a maioria dos tipos de redes esparsas, mesmo além das expectativas dos autores.

Como a complexidade do algoritmo *hipr* usado em cada região é $O(V^2\sqrt{E})$, onde V é o número de vértices e E é o número de arestas, o tempo de computação depende principalmente de V, não E. A rede foi particionada em várias regiões com números aproximados de vértices pelo BFS (Busca em Largura) começando pelo sorvedouro. As arestas que conectam duas regiões são tidas como arestas de fronteira e os vértices das arestas de fronteira são definidos como vértices de fronteira.

Três passos são executados iterativamente para empurrar fluxo para o sorvedouro até que não existam mais vértices ativos, ou seja, com a quantidade de fluxo entrando no vértice maior do que de fluxo saindo, em todas as regiões exceto o vértice fonte e o sorvedouro.

1. Rotular vértices de fronteira.

   Existem três tipos de rótulos dos vértices de fronteira para escolher, o tipo-I e o tipo-II têm possibilidade de empurrar o fluxo para t, enquanto o tipo-III não tem nenhuma possibilidade. O método de rotular os vértices de fronteira é o seguinte: se fora da região não existirem arestas de fronteira viáveis para o vértice v_1 empurrar o fluxo para o vértice v_2 (v_2 pertence à região diferente e o rótulo de v_2 não é tipo-III), então o rótulo de v_1 é definido como tipo-III. Distingui-se o tipo-I do tipo-II pela distância dos vértices de fronteira para t, o rótulo do vértice v_1 com menor distância é definido como tipo-I, senão como tipo-II.

   Como na Figura 3.3, o rótulo do vértice v_1 será definido como tipo-III, v_7 como tipo-I e v_8 como tipo-II, respectivamente.

2. Empurrar fluxo de dentro da região para a fronteira.

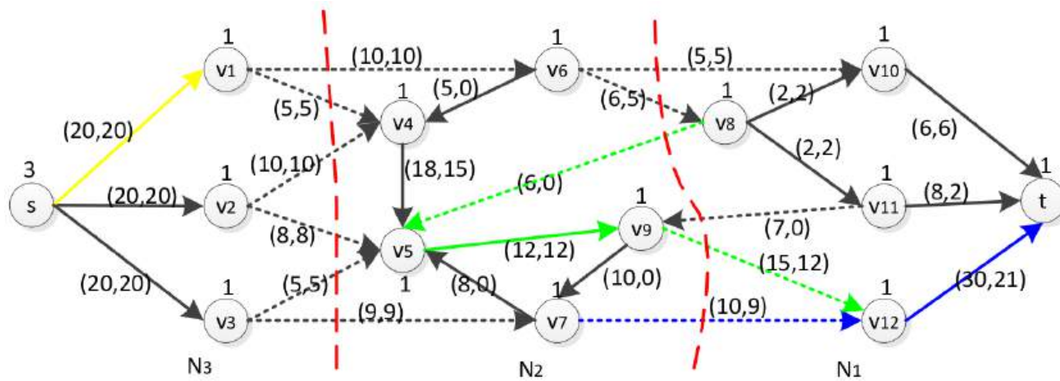


Figura 3.3: O modo rotular os vértices de fronteira.

O algoritmo *hipr* é utilizado para empurrar o fluxo de dentro da região para sua fronteira, mas o recurso do nosso método é empurrar o fluxo hierarquicamente, ou seja, o fluxo é empurrado para os vértices tipo-I, então para os vértices tipo-II e, por último, para os vértices tipo-III.

3. Descarregar fluxo para fora da região.

   O método de descarregar o fluxo dos vértices de fronteira para fora da região é semelhante ao modo de operação de rotular os vértices de fronteira. O fluxo dos vértices de fronteira de tipo-I e tipo-II são empurrados pelos arestas de fronteira de saída encontrados na operação de rotular os vértices de fronteira. Empurra-se o fluxo dos vértices tipo-II pelas arestas de fronteira de entrada.

Para reduzir o custo da passagem de mensagem, alguns caminhos viáveis de cruzamento de fronteira são ignorados. Então é necessário julgar se a solução ótima foi obtida através do BFS a partir de t. Se não, todas as regiões são juntadas em uma rede inteira com a ajuda do MPI e então o trabalho restante será feito em sequência.

O algoritmo MPI paralelo foi testado em uma *workstation* gráfica, que tem quatro processadores Quad-Core 64 bit que funcionam a 2.4 GHz com 64 GB de memória. O Visual Studio 2010 foi utilizado para codificar e gerar o arquivo executável. Quatro tipos de redes, que foram usadas no primeiro Desafio de Implementação DIMACS e usados por muitos algoritmos de fluxo máximo existentes, foram aplicados para testar o desempenho.

Desta forma, Jincheng e Lixin propuseram um novo algoritmo distribuído com MPI para o problema do fluxo máximo para grafos esparsos de larga escala. Vários métodos efetivos são utilizados para reduzir a troca de mensagens e é acrescentado cálculo adicional para certificar-se que a solução ótima pode ser obtida.

Este algoritmo é de grande importância para muitas aplicações de transporte e utilidade baseados em GIS (*Geographic Information Systems*). Especialmente para situação de emergência, o algoritmo paralelo cumpre um importante papel para agendamento de recursos.

CAPÍTULO
4

Algoritmo híbrido OpenMP-GPU

Ao longo deste estudo foram abordados alguns algoritmos sequenciais e paralelos para o problema do fluxo máximo, além das arquiteturas utilizadas para suas implementações. Existe um modelo de programação paralela que ainda não foi tratado que é a híbrido com GPU e OpenMP. Neste modelo, é possível alternar a execução do algoritmo entre a GPU e o processamento *multi-core* com OpenMP.

A proposta é desenvolver e implementar um algoritmo para o problema do fluxo máximo em rede utilizando este modelo, distribuindo o processamento, conforme a execução do algoritmo, entre a GPU e a CPU, e através do OpenMP, paralelizar algumas etapas do algoritmo em CPU, em especial as heurísticas de rotulação global e rotulação *gap*.

Neste capitulo serão descritos o algoritmo proposto, as técnicas utilizadas e os resultados obtidos, comparando-o com a melhor implementação do algoritmo *push-relabel* serial conhecida, a *hipr*.

4.1 Algoritmo híbrido OpenMP-GPU

O algoritmo proposto utiliza a abordagem *push-relabel* e alterna processamento *multicore* em CPU e em GPU, conforme o número de vértices ativos. O Algoritmo 9 mostra uma visão geral da abordagem híbrida OpenMP-GPU.

O bloco principal do algoritmo é executado sequencialmente por uma única *thread*, que posteriormente controla a execução de etapas paralelas. Primeiramente, executa-se a rotulação global paralela, que faz parte da inicialização dos dados. Em segundo lugar, executa-se a função de inicialização, preparando a variáveis h, e e c_f, que representam a distância, excesso do vértice

e capacidade residual da aresta, respectivamente. Além disso, inicializa-se a variável *ExcessTotal* com a soma das capacidades das arestas adjacentes ao vértice fonte s. Em seguida, executa-se o laço principal do algoritmo enquanto houver vértices ativos. Neste laço, o algoritmo decide pela execução das operações *push-relabel* na CPU ou GPU, de acordo com o número de vértices ativos, e copia os dados da CPU para a GPU e vice-versa, conforme necessário. Ao fim de cada iteração do laço principal, executa-se a função de rotulação global paralela.

Algoritmo 9 Algoritmo híbrido OpenMP-GPU

```
1:  parallel-global-labeling()
2:  initialize()
3:  LastDevice ← CPU
4:  while e(s) + e(t) < ExcessTotal do
5:      if NumActiveVertex < Threshold then
6:          pushrelabel-cpu()
7:          LastDevice ← CPU
8:      else
9:          if LastDevice = CPU then
10:             copy e, c_f from CPU to CUDA device
11:         end if
12:         copy h from CPU to CUDA device
13:         pushrelabel-gpu()
14:         copy e, c_f from CUDA device to CPU
15:         LastDevice ← GPU
16:     end if
17:     parallel-global-labeling()
18: end while
```

O Algoritmo 10 detalha os passos iniciais do algoritmo, com a inicialização das estruturas de dados. O Algoritmo 11 detalha a função *pushrelabel-cpu*, que é basicamente a mesma do algoritmo *push-relabel* de Goldberg, mas com alguns ajustes no momento de executar a rotulação global, como na linha 9 do algoritmo 11, em que k é o número de adjacentes visitados na rotulação dos vértices, e nm é o número de vértices multiplicado pelo número de arestas.

Nas próxima seções serão descritas as técnicas utilizadas e os resultados obtidos após sua utilização.

4.1.1 Operação pushrelabel-gpu

Em [16], na operação *pushrelabel-kernel*, o vértice v empurra fluxo para o vértice vizinho u que tenha a menor distância e que a aresta (v,u) não esteja saturada. Se não houver vizinho com uma distância $d(u)$ menor do que $d(v)$, então é feita a rotulação de v para a distância do menor vizinho de v mais uma unidade, ou seja, $d(v) = min(d(u) + 1 | u \in \texttt{Adj}(v))$, sendo que $\texttt{Adj}(v) = \{u | (v,u) \in A_f\}$.

A proposta no algoritmo *hipr* é empurrar fluxo para qualquer vértice em que $d(v) < d(u)$ e $(v,u) \in A_f$ e, se após a tentativa de empurrar fluxo para todos os vizinhos de v ainda houver fluxo excedente, ou seja, $e(v) > 0$, então é feita a rotulação de v conforme descrito no parágrafo anterior. Após a rotulação de v, tenta-se novamente empurrar fluxo para os vizinhos de v, começando com o que tinha a menor distância, descoberto na rotulação. Assim, não é necessário percorrer todos os vizinhos novamente após a rotulação.

Utilizou-se essa abordagem na etapa do algoritmo que é executada em CUDA, com a diferença de que a descoberta do vizinho de menor distância no grafo residual é feita no mesmo laço da operação de empurrar fluxo. Desta forma, utilizou-se apenas um laço e não dois como anteriormente. O Algoritimo 12 descreve a função *pushrelabel-gpu*. Como comentado na linha 1, cada vértice do grafo é operado por uma única *thread* CUDA.

Algoritmo 10 Descrição da função initialize

1: *parallel-global-update()*
2: for each vertex, sort its adjacency by h
3: $h(s) \leftarrow |V|$
4: $e(s) \leftarrow 0$
5: **for all** $u \in V - \{s\}$ **do**
6: $h(u) \leftarrow 1$
7: $e(u) \leftarrow 0$
8: **end for**
9: **for all** $(u,v) \in E$ **do**
10: $c_f(u,v) \leftarrow c(u,v)$
11: $c_f(v,u) \leftarrow c(v,u)$
12: **end for**
13: **for all** $(s,u) \in E$ **do**
14: $c_f(s,u) \leftarrow c_f(s,u) - c(s,u)$
15: $c_f(u,s) \leftarrow c_f(u,s) + c(s,u)$
16: $e(u) \leftarrow c(s,u)$
17: $ExcessTotal \leftarrow ExcessTotal + c(s,u)$
18: **end for**

4.1.2 Sobreposição de kernels em CUDA

Uma técnica para para aumentar desempenho de programas em CUDA é utilizar a sobreposição de *kernels*. Nesta técnica, substitui-se uma chamada de *kernel* em CUDA, que operaria sobre um conjunto de dados, por várias chamadas de *kernel* que operam sobre parte deste conjunto de dados. Cada chamada de *kernel* e feita utilizando-se um *stream* diferente. Consequentemente, a execução dos *kernels* ocorre concorrentemente.

Ao utilizar a sobreposição de *kernels*, deve-se evitar que ocorram operações sobre os mesmos dados, mesmo que seja em alguns momentos. Com esse

Algoritmo 11 Descrição da função pushrelabel-cpu

1: **while** $aMax \geq aMin$ **do**
2: $bucketMax \leftarrow bucketActive(aMax)$
3: $v \leftarrow$ first element of $bucketMax$
4: **if** $v \notin bucketMax$ **then**
5: $aMax \leftarrow aMax - 1$
6: **else**
7: remove v from $bucketMax$
8: **discharge**(v)
9: **if** $k > V.E$ **then**
10: **parallel-global-update()**
11: **end if**
12: **end if**
13: **end while**

cuidado, evita-se a ocorrência de *deadlocks*.

Foram testadas combinações diferentes para sobreposição de *kernels* e a que permitiu ganhos de performance foi o formato com 4 chamadas de *kernel*, cada uma operando sobre 1/4 do vetor de vértices.

4.1.3 Utilização de loop unrolling

Conforme [5], *loop unrolling* é uma técnica que tenta otimizar a execução de laços através da redução das instruções de manutenção de *loops*. Com esta técnica, ao invés de escrever uma vez o corpo de um laço e executá-lo repetidamente, o corpo é escrito no código múltiplas vezes. Qualquer laço interno tem suas iterações reduzidas ou é completamente removido. *Loop unrolling* é a técnica mais efetiva ao melhorar o desempenho de laços que processam *arrays* sequencialmente, onde o número de iterações é conhecido antes da execução do laço. Considere o seguinte fragmento abaixo:

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

Repetindo-se o corpo do laço uma vez, o número de iterações pode ser reduzido para a metade comparado ao laço original:

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

A razão para os ganhos de desempenho através do *loop unrolling* pode não ser facilmente percebida apenas olhando o código de alto nível. Os ganhos vêm das melhorias de instrução de baixo nível e otimizações que o compila-

dor executa com a técnica. Por exemplo, no código acima, a condição $i < 100$ é conferida apenas cinquenta vezes, comparada às 100 conferências do laço original. Adicionalmente, devido as leituras e escritas executadas em cada instrução serem independentes, as operações de memória podem ser realizadas simultaneamente pela CPU.

Unrolling em CUDA pode significar uma variedade de coisas. Entretanto, o objetivo é sempre o mesmo: melhorar o desempenho reduzindo as sobrecargas de instrução e criando mais instruções independentes para agendar. Como resultado, mais operações concorrentes são adicionadas ao *pipeline*, conduzindo a uma saturação maior de instruções e largura de banda de memória. Isto dá ao gerenciador de *warps* mais *warps* elegíveis, que podem ajudar a esconder latência de memória e instrução.

Utilizou-se esta técnica internamente na operação *pushrelabek_kernel*. Para isso, cada *thread* CUDA opera sobre mais de um vértice através de um laço e a instrução `#pragma unroll` é usada pra sugerir ao compilador CUDA que este laço deve ser desenrolado automaticamente. Esta mesma instrução foi utilizada no laço que itera sobre as arestas adjacentes para empurrar fluxo.

Algoritmo 12 Descrição da função pushrelabel-gpu

1: **while** $e(u) > 0$ and $h(u) < |V|$ **do** ▷ Nó u operado pela *thread* u
2: $e' \leftarrow e(u)$
3: $h' \leftarrow \infty$
4: **for all** $(u,v) \in E_f$ **do**
5: $h'' \leftarrow h(v)$
6: **if** $h'' < h'$ **then**
7: $h' \leftarrow h''$
8: **end if**
9: **if** $h(u) > h'$ **then**
10: $d \leftarrow$ min(e', $c_f(u,v)$)
11: AtomicAdd($c_f(v,u)$, d)
12: AtomicSub($c_f(u,v)$, d)
13: AtomicAdd($e(v)$, d)
14: AtomicSub($e(u)$, d)
15: **end if**
16: **end for**
17: **if** $e(u) > 0$ and $h' < V$ **then**
18: $h(u) \leftarrow h' + 1$
19: **else**
20: **break**
21: **end if**
22: **end while**

4.1.4 Paralelização da rotulação global utilizando OpenMP

OpenMP é um modelo de programação paralela em CPU que utiliza diretivas de compilador para identificar regiões paralelas. Os compiladores que suportam diretivas OpenMP podem usá-las como sugestões do programador em como paralelizar uma aplicação. Com bem pouco código, o paralelismo em *multi-core* no *host* pode ser alcançado. Pode-se também utilizar CUDA e aproveitar o OpenMP para melhorar não apenas a portabilidade e a produtividade, mas também a performance do código no *host*. Como alternativa ao uso de um laço para executar vários *kernels* utilizando *streams* diferentes, podem-se utilizar as diretivas do OpenMP.

Conforme estudado em [16], implementar a rotulação global utilizando CUDA piorou o *speedup*, então os autores mantiveram a implementação serial na CPU. Em uma tentativa de melhorar o tempo da rotulação global utilizou-se o OpenMP e baseando-se nas proposta de [22] para o BFS, alcançou-se um *speedup* de até 3 a 8 vezes na rotulação global, conforme o tipo de grafo. Consequentemente, o tempo total de execução do algoritmo para o fluxo máximo foi reduzido.

No Algoritmo 13 é descrita a rotulação paralela. A variável $currentFrontier$ define a fronteira atual, uma lista com até n elementos. Já $localFrontiers$ é um vetor de fronteiras locais e $numThread$ é o número da *thread* atual. Utiliza-se o termo "fronteira" para definir o conjunto de vértices no nível atual da busca em largura. Ao visitar todos os vértices da fronteira atual paralelamente, produz-se a próxima fronteira, primeiramente através de fronteiras locais, uma por *thread*. Posteriormente, as fronteiras locais são fundidas em uma única fronteira, que passa a ser a fronteira atual a ser visitada e a distância é incrementada. Para a fusão das fronteiras locais, utiliza-se a soma prefixada paralela para encontrar a posição de cada fronteira local no conjunto resultante. Esta parte do algoritmo termina quando não há mais vértices a serem visitados.

O próximo passo é marcar todos os vértices que não foram visitados na busca em largura e remover o excesso de fluxo deste vértice da variável *ExcessTotal*, que controla o resultado final. É a mesma abordagem utilizada em [16].

4.1.5 Paralelização da rotulação gap utilizando OpenMP

Assim como a rotulação global, a rotulação *gap* melhora substancialmente o tempo do algoritmo *push-relabel*. Entretanto, esta heurística só é vantajosa para o algoritmo *hipr* executado na CPU. Isto acontece porque o algoritmo *hipr* mantém listas de vértices ativos, uma para cada distância possível no grafo. Em CUDA, as implementações de listas não apresentam bom tempo de

Algoritmo 13 Rotulação global paralela

```
1: add n − 1 in currentFrontier
2: d ← 0
3: for all u ∈ V do in parallel
4:     h(u) ← n
5: end for
6: while currentFrontier ≠ ∅ do
7:     for all u ∈ currentFrontier do in parallel
8:         for all c_f(v,u) ∈ E_f do
9:             if h(v) = n then
10:                h(v) ← d
11:                add v in localFrontiers(numThread)
12:            end if
13:        end for
14:    end for
15:    merge localFrontiers in currentFrontier
16:    d ← d + 1
17: end while
18: for all u ∈ V do
19:    if u is not labeled ∧ u is not marked then
20:        mark u
21:        ExcessTotal ← ExcessTotal − e(u)
22:    end if
23: end for
```

execução.

Quando uma lista fica vazia, a rotulação *gap* é executada conforme o Algoritmo 14, buscando-se todos os vértices ativos nas listas com distância maior do que a da lista vazia atual, removendo-se estes vértices da lista e definindo a nova distância como o número de vértices. Como a rotulação dos vértices nessas listas é independente, utilizou-se o OpenMP para executá-las paralelamente. No algoritmo, *buckets* são as listas de vértices, conforme a distância. Já *emptyBucket* é a lista vazia atual.

Algoritmo 14 Rotulação gap paralela

```
1: for all b ∈ buckets ∧ b > emptyBucket ∧ b ≤ buckets(maxDistance) do in parallel
2:     for all v ∈ b do in parallel
3:         h(v) ← n
4:     end for
5:     b ← ∅
6: end for
```

4.1.6 Outros detalhes de implementação

Da mesma forma que [16], o algoritmo decide, conforme o número de vértices ativos, se a execução será realizada pela CPU ou pela GPU. Isso permite

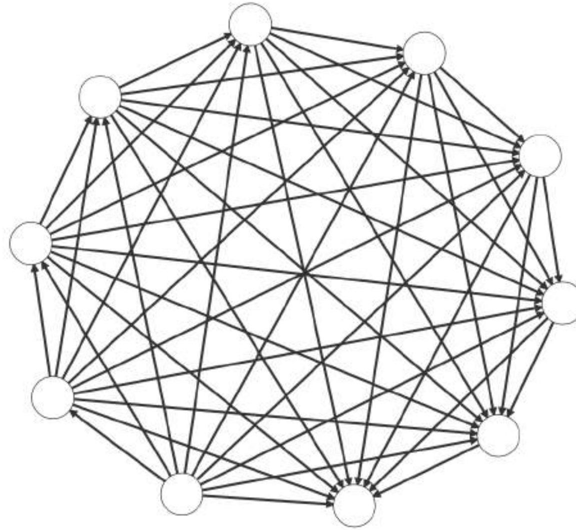


Figura 4.1: Um exemplo de grafo do tipo Acyclic Dense.

que a execução ocorra onde terá o melhor tempo de execução naquele momento. Esse limite, chamado de *threshold*, varia conforme o desempenho da GPU utilizada. Quanto maior o desempenho da GPU, menor o valor do *threshold* e vice-versa. Esse valor é parametrizado, ou seja, deve ser predefinido e tem o valor fixo durante a execução do algoritmo.

Uma melhoria foi alcançada na estrutura de dados utilizada para armazenar as arestas adjacentes de cada vértice. No início do algoritmo, é executada uma rotulação global e posteriormente as arestas adjacentes de cada vértice são ordenadas em ordem decrescente das distâncias em relação ao sorvedouro. Entretanto, após esta ordenação, as distâncias de cada vértice são redefinidas para 1, da mesma forma que no algoritmo *hipr*. A função de inicialização, como vista no Algoritmo 10, mostra estes passos.

4.2 Resultados computacionais

Após a utilização das melhorias foram feitos testes comparando a versão em CUDA melhorada, a qual chamaremos *openmp-cuda*, com a melhor versão serial existente do *push-relabel*, a *hipr*.

Os testes foram realizados na *Amazon Elastic Compute Cloud* (*Amazon* EC2), que é um serviço web da *Amazon* para computação em nuvem. Ele fornece instâncias, que são máquinas virtuais, para disponibilizar processamento em CPU e GPU sob demanda. A instância utilizada foi do tipo g2.2xlarge, que disponibiliza um processador Intel Xeon E5-2670, com 8 vCPUs, 15GB de memória RAM e uma placa gráfica Nvidia Grid K520, com 1536 núcleos CUDA e 4GB de memória de vídeo.

Testou-se 5 tipos de grafos de entrada utilizados no 1º DIMACS *Implemen-*

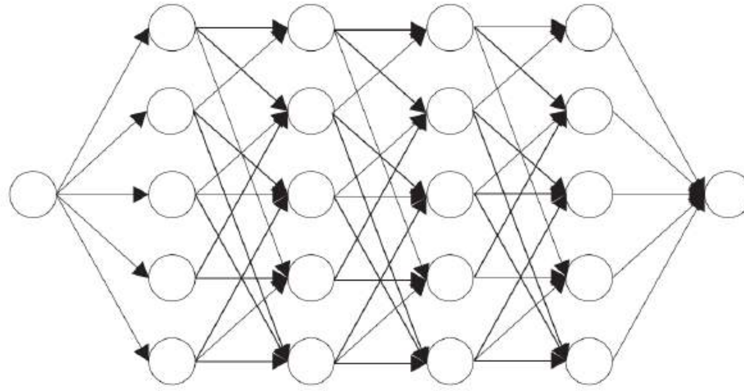


Figura 4.2: Um exemplo de grafo do tipo Washington-RLG.

tation Challange [19]:

1) **Acyclic-Dense**: estes são grafos do tipo denso acíclico direcionado completo, em que cada vértice está conectado a cada outro vértice. Foram testados grafos de 2000, 4000, e 6000 vértices.

2) **Genrmf-long**: este grafo é composto de l_1 grades quadradas de vértices (quadros) cada um contendo $l_2 \times l_2$ vértices. O vértice fonte fica em um canto do primeiro quadro e o sorvedouro no canto oposto do último quadro. Cada vértice é conectado com seus vizinhos de grade e com um vértice escolhido aleatoriamente no próximo quadro. Testou-se grafos com $l_1 = 192$, $l_2 = 24$ (110592 vértices e 533952 arestas), $l_1 = 224$, $l_2 = 28$ (175616 vértices e 852208 arestas) e $l_1 = 256$, $l_2 = 32$ (262144 vértices e 1276928).

3) **Genrmf-wide**: a topologia é a mesma dos grafos Genrmf-long, exceto os valores de l_1 e l_2. Os quadros são maiores nos grafos Genrmf-wide do que nos grafos Genrmf-long. Testaram-se grafos com l_1 = 48 e l_2 = 48 (46656 vértices e 226800 arestas), l_1 = 64 e l_2 = 64 (262144 vértices e 1290240 arestas), l_1 = 96 e l_2 = 96 (884736 vértices e 4377600 arestas) e l_1 = 128 e l_2 = 128 (2097152 vértices e 10403840 arestas).

4) **Washington-RLG-long**: estes grafos são grades retangulares de vértices com w linhas e l colunas. Cada vértice em uma linha tem três arestas conectando vértices aleatórios na próxima linha. A fonte e o sorvedouro são externos à grade, a fonte tem as arestas com todos os vértices na linha mais acima e todos os vértices na linha mais abaixo têm arestas com o sorvedouro. Testaram-se grafos de w = 512, l = 1024 (524290 vértices e 1572352 arestas), w = 768, l = 1280 (983042 vértices e 2948352 arestas) e w = 512, l = 1024 (1572866 vértices e 4717568 arestas).

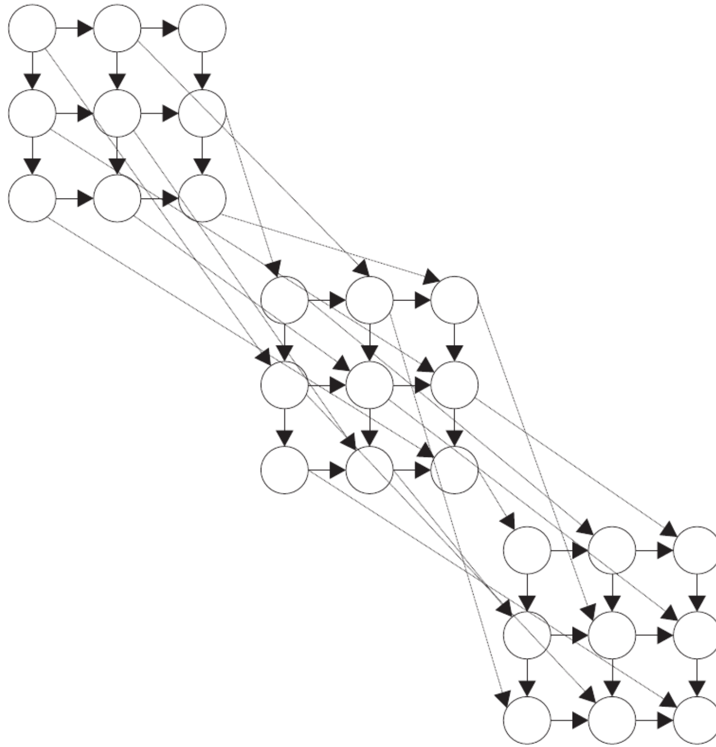


Figura 4.3: Um exemplo de grafo do tipo Genrmf.

Vértices	**hipr**	**openmp-cuda**	***speedup***
2000	0,25	0,13	1,92
4000	0,78	0,33	2,36
6000	3,21	1,55	2,07

Tabela 4.1: Tempos(s) e *speedups* para grafos Acyclic-Dense.

5) **Washington-RLG-wide**: o mesmo para os grafos Washington-RLG-long exceto para os valores de w e l. Cada linha nos grafos Washington-RLG-long é mais longa. Testaram-se grafos com w = 512, l = 512 (262146 vértices e 785920 arestas), w = 768, l = 768 (589826 vértices e 1768704 arestas) e w = 1024, l = 1024 (1048578 vértices e 3144704 arestas).

A figura 4.1 mostra um exemplo de grafo do tipo Acyclic Dense, a figura 4.2 mostra um exemplo de grafo do tipo Washington-RLG e a figura 4.3 mostra um exemplo de grafo do tipo RMF. Para cada tipo de grafo, foram geradas 5 instâncias, utilizando sementes diferentes para o gerador pseudoaleatório. Cada instância foi testada 3 vezes e foi calculada a média do tempo de execução.

Para os grafos do tipo *Acyclic-Dense*, como visto na Figura 4.1, o algoritmo *openmp-cuda* teve um tempo menor de execução do que o *hipr* para grafos do tipo Genrmf-long em todos os casos, alcançando *speedup* de 2,07 para 6.000 vértices.

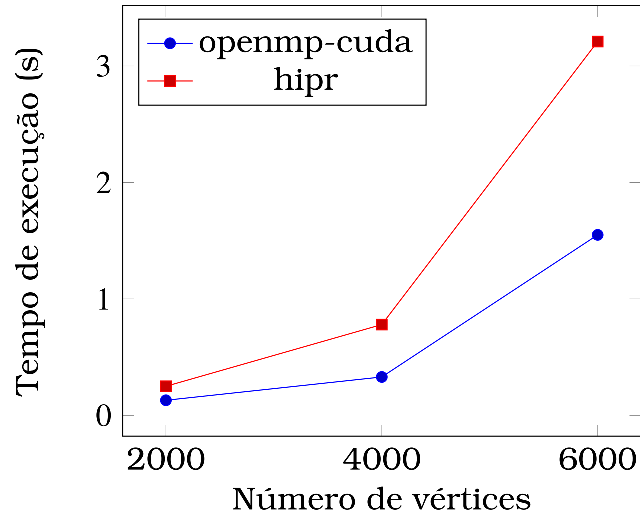


Figura 4.4: Resultados computacionais em grafos Acyclic-Dense.

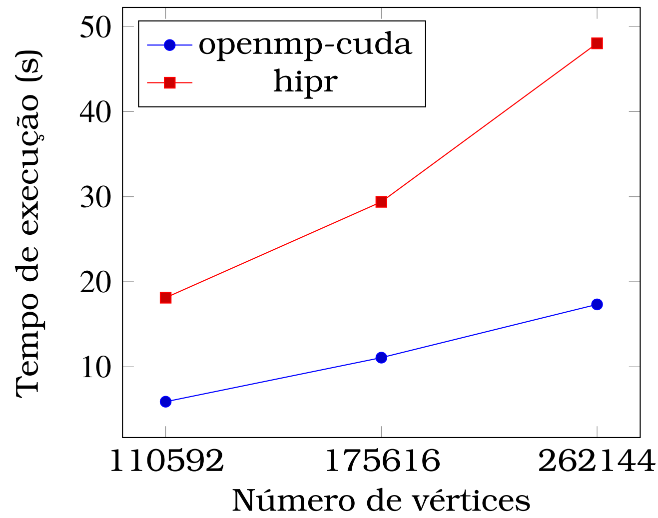


Figura 4.5: Resultados computacionais em grafos Genrmf-long.

Vértices	**hipr**	**openmp-cuda**	***speedup***
110592	18,12	5,89	3,07
175616	29,39	11,07	2,65
262144	46,88	17,26	2,71

Tabela 4.2: Tempos(s) e *speedups* para grafos Genrmf-long.

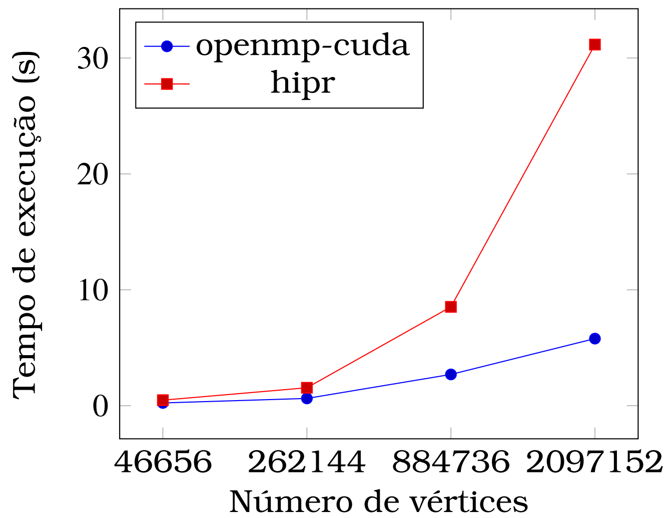


Figura 4.6: Resultados computacionais em grafos Genrmf-wide.

Vértices	**hipr**	**openmp-cuda**	***speedup***
46656	0,48	0,24	1,95
262144	1,55	0,63	2,46
884736	8,53	2,70	3,15
2097152	31,16	5,79	5,38

Tabela 4.3: Tempos(s) e *speedups* para grafos Genrmf-wide.

Conforme mostra a Figura 4.5, o algoritmo *openmp-cuda* teve um tempo menor de execução do que o *hipr* para grafos do tipo Genrmf-long em todos os casos, alcançando um *speedup* de 2,77 para 262.144 vértices.

Para o grafo do tipo Genrmf-wide, como visto na Figura 4.6, o algoritmo *openmp-cuda* teve um tempo de execução menor do que o *hipr* para todos os tamanhos de grafos, alcançando o *speedup* de 5,38 para 2.097.152 vértices.

No caso dos grafos do tipo Washington-RLG-long, conforme a Figura 4.7, o algoritmo *openmp-cuda* conseguiu um tempo de execução menor do que *hipr* em todos os casos, alcançando *speedup* de 3,29 para 1.572.866 vértices.

Por fim, no caso dos grafos do tipo Washington-RLG-wide, o algoritmo *openmp-cuda* teve o tempo de execução menor em todos os casos, com *speedup* de até 2,60 para 1.048,578 vértices.

Vértices	**hipr**	**openmp-cuda**	***speedup***
524290	1,11	0,57	1,92
983042	3,30	1,20	2,75
1572866	6,35	1,93	3,28

Tabela 4.4: Tempos(s) e *speedups* para grafos Washington-RLG-long.

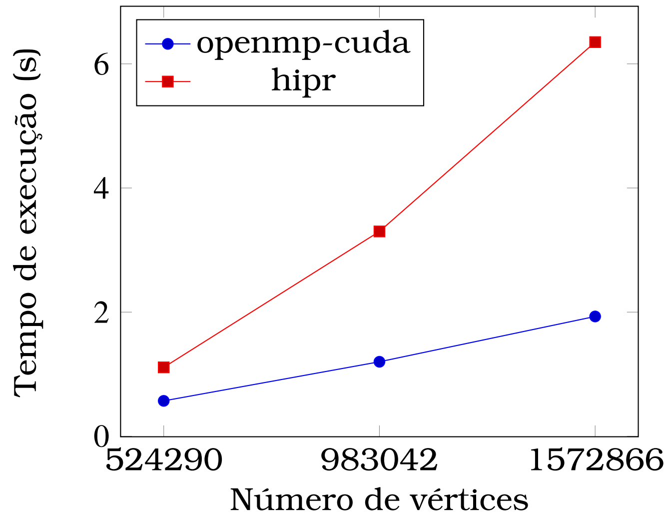


Figura 4.7: Resultados computacionais em grafos Washington-RLG-long.

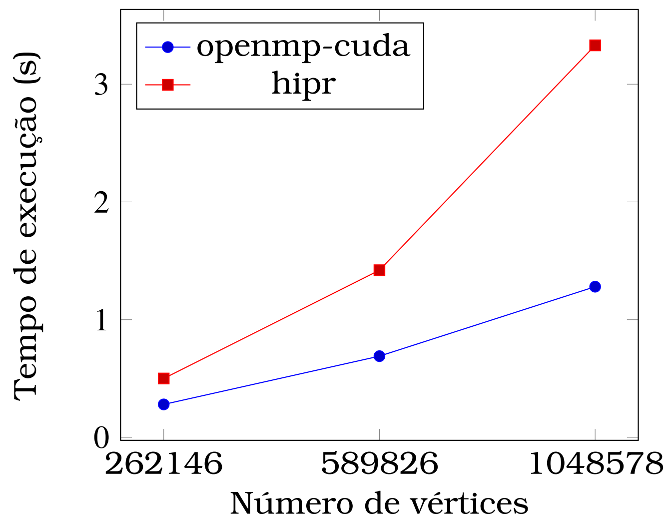


Figura 4.8: Resultados computacionais em grafos Washington-RLG-wide.

Vértices	**hipr**	**openmp-cuda**	***speedup***
262146	0,50	0,28	1,75
589826	1,42	0,69	2,05
1048578	3,33	1,22	2,71

Tabela 4.5: Tempos(s) e *speedups* para grafos Washington-RLG-wide.

De modo geral, houve ganhos de *speedup* para todos os tipos de grafos escolhidos. Entretanto, a fonte dos ganhos variaram conforme o tipo de grafo. Por exemplo, os grafos do tipo Genrmf se beneficiaram da paralelização da rotulação global, mas não da paralelização da rotulação *gap*, que raramente é executada para este tipo de grafo. Além disso, houve boa parte da execução em GPU, sendo que no ambiente testado, o momento ideal para transferir a execução para GPU é quando o número de vértices ativos supera 17.000 vértices.

Por outro lado, os grafos do tipo Washington-RLG se beneficiaram da paralelização da rotulação *gap*, que é executada frequentemente para este tipo de grafo. Além disso, como o número de vértices ativos não atinge 17.000, a execução em GPU não é vantajosa, permanecendo completamente na CPU.

Capítulo 5

Conclusão

Neste trabalho estudou-se o problema do fluxo máximo em redes através da abordagem paralela, numa abordagem híbrida com OpenMP e CUDA. Primeiramente estudou-se algumas arquiteturas paralelas atuais, como *PRAM*, memória distribuída com MPI, memória compartilhada com OpenMP e em GPUs Nvidia com CUDA. Em segundo lugar, estudou-se as implementações paralelas para essas arquiteturas, como a abordagem livre de *lock multithread* e em CUDA, a abordagem auto-estabilizante em CUDA e a baseada em *push-relabel* em MPI.

Desenvolveu-se a implementação de um algoritmo para fluxo máximo em redes que distribui dinamicamente o processamento entre *threads* na CPU, através do OpenMP, e GPU através do CUDA. Além disso, as heurísticas da rotulação global e rotulação *gap* foram paralelizadas através das diretivas do OpenMP, uma abordagem ainda não utilizada por outros autores. Consequentemente, estas implementações paralelas das heurísticas podem ser utilizadas por outros algoritmos baseados em *push-relabel*, já que são independentes da execução principal do algoritmo. Após a comparação com o melhor algoritmo sequencial existente, o *hipr*, houve ganhos para todos os tipos de grafos estudados, alcançando-se *speedups* de 1,75 até 5,38.

O algoritmo pode ser melhorado explorando-se funcionalidades das GPUs Nvidia mais modernas, como o paralelismo dinâmico e o *warp shuffle*, além de outras técnicas como os *warps* virtuais dinâmicos. Dessa forma o *threshold* pode ser diminuído e mais tipos de grafos podem se beneficiar da execução em GPU.

Referências Bibliográficas

[1] R. Anderson and J. C. Setubal. A parallel implementation of the push-relabel algorithm for the maximum flow problem. *J. Parallel Distrib. Comput.*, 29(1):17–26, Aug. 1995. Citado nas páginas 2 e 13.

[2] D. Bader and V. Sachdeva. A cache-aware parallel implementation of the push-relabel network flow algorithm and experimental evaluation of the gap relabeling heuristic. In *Proc. of the 18th ISCA International Conference on Parallel and Distributed Computing Systems,*, 2005. Citado na página 2.

[3] G. C. Caragea and U. Vishkin. Better speedups for parallel max-flow. In *SPAA*, 2011. Citado nas páginas 2 e 25.

[4] B. Chapman, G. Jost, and R. van van der Pas. *Using OpenMP: Portable Shared Memory Parallel Programming (Scientific and Engineering Computation)*. The MIT Press, scientific and engineering computation edition, 10 2007. Citado na página 7.

[5] J. Cheng, M. Grossman, and T. McKercher. *Professional CUDA C Programming*. Wrox, 2014. Citado na página 32.

[6] S. Cook. *CUDA Programming: A Developer's Guide to Parallel Computing with GPUs*. Elsevier Science, 2012. Citado na página 10.

[7] H. C. da Silva. Uma implementação do algoritmo auto-estabilizante para o problema do fluxo máximo usando CUDA. Master's thesis, UFMS, Mato grosso do Sul, Brazil, 2013. Citado na página 22.

[8] E. A. Dinic. Algorithm for solution of problem of maximum flow in networks with power estimation. *Soviet Mathematics Doklady*, 11:1277–1280, 1970. Citado na página 2.

[9] J. Edmonds and R. M. Karp. Theorical improvements in the algorithmic efficiency for network flow problems. *Journal of the ACM*, 19:248–264, 1972. Citado na página 2.

[10] D. R. Ford and D. R. Fulkerson. *Flows in Networks*. Princeton University Press, Princeton, NJ, USA, 1962. Citado na página 2.

[11] A. V. Goldberg. Processor-efficient implementation of a maximum flow algorithm. *Information Processing Letters*, 38:179–185, 1991. Citado na página 2.

[12] A. V. Goldberg and R. E. Tarjan. A new approach to the maximum flow problem. In *Proceedings of the eighteenth annual ACM Symposium on Theory of Computing*, pages 136–146, 1986. Citado nas páginas 2 e 13.

[13] S. Gosh, A. Gupta, and S. V. Pemmaraju. A self-stabilizing algorithm for the maximum flow problem. In *Conference Proceedings of the 1995 IEEE Fourteenth Annual International Phoenix Conference on Computers and Communications*, pages 8–14, 1995. Citado na página 22.

[14] Z. He and B. Hong. Asynchronous multi-threaded algorithm for the max-flow/min-cut problem with non-blocking global relabeling heuristic. *IEEE Transactions on Parallel and Distributed Systems*, 2011. Citado nas páginas 16, 21, e 22.

[15] B. Hong. A lock-free multi-threaded algorithm for the maximum flow problem. In *IEEE International Parallel and Distributed Processing Symposium*, pages 1–8, 2008. Citado nas páginas 2 e 13.

[16] B. Hong and Z. He. An asynchronous multithreaded algorithm for the maximum network flow problem with nonblocking global relabeling heuristic. volume 22, pages 1025–1033, Los Alamitos, CA, USA, 2011. IEEE Computer Society. Citado nas páginas 3, 19, 30, 34, e 35.

[17] M. Hussein, A. Varshney, and L. Davis. On implementing graph cuts on CUDA. In *First Workshop on General Purpose Processing on Graphics Processing Units*, Boston, MA, October 2007. Citado na página 18.

[18] J. Jincheng and W. Lixin. A MPI parallel algorithm for the maximum flow problem. In *Proceedings of the 12th International Conference on GeoComputation*, 2013. Citado na página 25.

[19] D. S. Johnson and C. C. McGeoch, editors. *Network Flows and Matching: First DIMACS Implementation Challenge*. American Mathematical Society, Boston, MA, USA, 1993. Citado na página 37.

[20] J. Jájá. *An Introduction to Parallel Algorithms*. Addison-Wesley, 1992. Citado na página 5.

[21] A. V. Karzanov. Determining the maximal flow in a network by the method of preflows. *Soviet Mathematics Doklady*, 15:434–437, 1974. Citado na página 2.

[22] E. Mastrostefano and M. Bernaschi. Efficient breadth first search on multi-GPU systems. *J. Parallel Distrib. Comput.*, 73(9):1292–1305, Sept. 2013. Citado na página 34.

[23] Y. Shiloach and U. Vishkin. An $O(n^2 \log n)$ parallel max-flow algorithm. *Journal of Algorithms*, 3:128–146, 1982. Citado nas páginas 2 e 25.

[24] S. Soner and C. Ozturan. Experiences with parallel multi-threaded network maximum flow algorithm. Technical report, Technical report, PRACE, 2013. Citado na página 14.

[25] V. Vineet and P. J. Narayanan. CUDA cuts: Fast graph cuts on the GPU. In *Computer Vision and Pattern Recognition Workshops. IEEE Computer Society*, pages 1–8. Citado na página 18.