DEPARTAMENTO
DE INFORMÁTICA

NOVA SCHOOL OF
SCIENCE & TECHNOLOGY

RICARDO JORGE REBELO PEREIRA HENRIQUES MARTINHO

Licenciado em Engenharia Informática

GRAFOS DINÂMICOS EM GPUS

MESTRADO EM ENGENHARIA INFORMÁTICA

Universidade NOVA de Lisboa
Novembro, 2021

NOVA SCHOOL OF SCIENCE & TECHNOLOGY

DEPARTAMENTO
DE INFORMÁTICA

GRAFOS DINÂMICOS EM GPUS

RICARDO JORGE REBELO PEREIRA HENRIQUES MARTINHO

Licenciado em Engenharia Informática

Orientador: Hervé Paulino
Professor Associado, Universidade NOVA de Lisboa

Júri:

Presidente: Doutor Pedro Abílio Duarte de Medeiros
Professor Associado, Universidade NOVA de Lisboa

Arguentes: Doutor Pedro Manuel Pinto Ribeiro
Professor Auxiliar, Faculdade de Ciências da Universidade do Porto

Doutor Hervé Miguel Cordeiro Paulino
Professor Associado, Universidade NOVA de Lisboa

MESTRADO EM ENGENHARIA INFORMÁTICA

Universidade NOVA de Lisboa
Novembro, 2021

Grafos Dinâmicos em GPUs

Agradecimentos

Em primeiro lugar gostava de agradecer ao meu professor e orientador Hervé Paulino, pelo contínuo apoio no decorrer desta longa tese. Esta tese não seria sido possível sem a sua ajuda.

Queria também agradecer à FCT/UNL por ter sido a minha segunda casa durante a minha instrução ao longo deste curso.

Um obrigado também a todos os meus amigos que me ajudaram incondicionalmente, que me ouviram e me deram força nos momentos em que mais precisava.

Por fim mas não menos importante à minha família que fez com que tudo isto tenha sido possível.

Resumo

Os grafos são estruturas de dados utilizadas para modelar diversos problemas em várias áreas da ciência, desde a sua representação de moléculas na química, ao uso da representação de redes de comunicação na ciência da computação. À medida que a sua popularidade aumenta, aumenta também a necessidade de aplicações que simultaneamente analisem e processem eficientemente grafos de grandes dimensões. Nestes casos a utilização das capacidades altamente paralelizáveis da GPUs tem se relevado promissora.

Muitos dos problemas modelados com grafos necessitam que o grafo evolua com o passar do tempo, com a adição e/ou remoção de novos nós e arestas. Este requisito propõe novos desafios no processamento de grafos no GPU tornando o desenvolvimento destas ferramentas uma área de investigação relevante e atual.

Nesta dissertação de mestrado apresentamos Marrow-Graph, uma biblioteca de processamento de grafos dinâmicos em GPU, construída usando o Marrow – uma framework em C++ para computação paralela de sistemas heterogéneos, desenvolvida no centro de investigação NOVA-LINCS. Os nossos testes revelam que com a partição do grafo em diferentes segmentos é possível criar um grafo que suporta adições e remoções de vértices e arestas eficientes, em qualquer estado. Tornando possível não só evitar sobrecargas de uma reconstrução do grafo como também é possível o seu processamento em GPU.

Palavras-chave: Grafos, GPU, Processamento de Grafos Dinâmicos, Marrow

Abstract

Graphs are data structures used to model diverse problems in various Sciences, from the representation of molecules in Chemistry, to their use in the representation of comunication networks in computer science. As their popularity increases, so does the necessity for applications that simultaneously analyze and process efficiently large scale graphs. In these cases, the capability of a GPU to perform highly parallel computation has shown promise.

Many of the problems modelled with graphs require the graph to evolve over time, with the addition and/or removal of new edges and vertices. This requirement poses new challenges in the processing of graphs by the GPU, turning the development of these tools into a relevant and current research area.

In this master's thesis we present Marrow-Graph, a dynamic GPU graph processing library, built using Marrow - a C++ framework for parallel computing of heterogeneous systems, developed at the NOVA-LINCS investigation center. Our tests show that by partiotioning the graph in different segments, it is possible to create a graph that supports the efficient addition and removal of vertices and edges in any state, making it possible not only to to avoid graph reconstruction overhead, but also possible to process it by GPU.

Keywords: Marrow, GPU, Graphs, Dynamic Graph Processing Libraries, Graph Processing Library

Índice

Índice de Figuras

Índice de Tabelas

Índice de Listagens

Siglas

API Application Programming Interface 8, 16
ASIC Application-Specific Integrated Circuit 2
AST Abstract Syntax Tree 21

BFS Breadth-First Search 55, 56, 58, 60, 63, 64, 67, 69
BSP Bulk Synchronous Parallel 16

CPU Unidade Central de Processamento 2, 3, 5, 6, 10, 14, 19, 20, 21, 22, 27, 29, 30, 46, 52, 60, 68
CSR Compact Sparse Row xi, 13, 14

DRAM Dynamic Random-Access Memory 6

FPGA Field Programmable Gate Arrays 2

GAS Gather Apply Scatter 15
GPS Sistema de Posicionamento Global 1
GPU Unidade de Processamento Gráfico xi, 1, 2, 3, 4, 5, 6, 7, 8, 10, 11, 12, 14, 15, 17, 18, 19, 20, 21, 22, 24, 27, 29, 37, 38, 40, 46, 47, 52, 53, 55, 56, 57, 68

SFU Special Functional Units 6
SM Streaming Multiprocessor xi, 6, 7, 8
SSSP Single Source Shortest Path 55, 56, 63, 67, 69

1

Introdução

1.1 Motivação

As estruturas de dados conhecidas por grafos apresentam-se omnipresentes em várias áreas da ciência. Como consequência da sua grande adoção, muitos problemas de interesse prático podem ser formulados como operações sobre grafos. Por exemplo, na área de biologia, os grafos são utilizados em simulações de vias biológicas, usando modelos estatísticos de expressão genética representados em grafos [17]. Na química, tipicamente a representação de interações entre átomos, criando moléculas, é feita com grafos. Na área de ciência da computação, os grafos são utilizados como estruturas de dados capazes de representar relacionamentos entre pessoas em redes sociais. Os grafos também são usados em simultâneo com Sistema de Posicionamento Global (GPS) para determinar o melhor trajeto entre duas localizações [31].

Esta crescente e contínua utilização de grafos em aplicações comerciais de modo a representar e analisar redes sociais, serviços financeiros e simulações científicas juntamente com a influencia de aplicações modernas da "*Big Data*", tem gerado grafos de grande escala com milhões de vértices e arestas [7, 29, 28]. Como consequência de tudo isto tem havido uma procura cada vez maior em acelerar as primitivas de grafos.

Como forma de diminuir o tempo necessário para processar grafos, desde há alguns anos têm surgido propostas que tiram partido de aceleradores de hardware de modo a ser possível mais rapidamente analisar grafos. Algumas das solução usadas como aceleradores de processamento de grafos são:

Unidades de Processamento Gráfico (GPUs) A GPU é um chip geralmente disponível em todos os computadores domésticos, sendo o seu uso principal na renderização de imagens. No contexto da computação a GPU é usado geralmente como um co-processador onde a sua eficiência energética e o grande números de núcleos tornam a GPU um candidato interessante em aplicações como o processamento de grafos que tiram partido das suas capacidades altamente paralelizáveis. Neste contexto a utilização da GPU denomina-se de *General-purpose computing on graphics processing*

units (GPGPU) onde a GPU é utilizada para o processamento de métodos tipicamente realizadas por Unidade Central de Processamento (CPU). Atualmente há várias bibliotecas de processamento de grafos que utilizam a GPU com acelerador [27].

Field Programmable Gate Arrayss (FPGAs) As FPGAs são chips desenhados com a capacidade de serem reprogramados depois de serem fabricados. As FPGAs e GPUs são os dois aceleradores programáveis mais comuns [6]. Tal como as GPUs a sua eficiência energética é apelativa tal como as suas capacidades paralelizáveis, no seu uso como aceleradores de processamento de grafos [10].

Application-Specific Integrated Circuits (ASICs) São chips desenhados para efetuar uma tarefa especifica, ao contrário dos FPGAs depois de fabricados não é possível alterar o seu comportamento. Tal como as GPUs e FPGAs apresentam capacidade altamente paralelizáveis, com tipicamente a melhor eficiência energética entre os três. No entanto a impossibilidade de serem reprogramados torna os ASICs pouco apelativos para um ambiente de desenvolvimento [10].

1.2 Definição do Problema

A crescente e constante mutação de informação é um problema quando os sistemas que processam esta informação não capturam a natureza dinâmicas destas alterações. Numa rede social as relações vão evoluindo, novas relações são criadas enquanto outras são apagadas, a evolução destas relações é permanente. Sistemas que não contabilizam estas alterações não representam realisticamente o próprio sistema que querem processar.

Apesar de existirem bibliotecas de processamento de grafos em GPU, como CuSha, MapGraph, Gunrock [30] estes não suportam a modificação dinâmica de grafos após o grafo ter sido carregado para a memória da GPU, fazendo com que qualquer alteração ao grafo cause uma retransmissão na integra para a GPU antes que seja possível voltar a processar o grafo.

No entanto esta abordagem nem sempre é realista. É possível imaginar que num grafo que esteja continuamente a sofrer alterações, sejam estas adições ou remoções de arestas ou vértices, a sobrecarga de reconstrução e retransmissão irá ser traduzida num atraso considerável até que o grafo possa ser avaliado com uso de diferentes algoritmos. Qualquer análise feita, rapidamente poderá ficar desatualizada e o seu resultado poderá já não se aplicar ao grafo atual.

Deste modo nota-se que há uma necessidade em haver mais bibliotecas de processamento de grafos dinâmicos de alto desempenho. Com cada vez mais informação a ser gerada é espectável assumir que esta necessidade irá aumentar. Para resolver este problema, neste trabalho propomo-nos a desenvolver uma biblioteca de processamento de grafos dinâmicos.

Para alcançar uma biblioteca para processamento de grafos dinâmica devemos responder algumas questões:

- É possível adicionar arestas e vértices eficientemente depois de o grafo ser carregado para a memória da GPU?
- Que tipo de estruturas de dados podem ser usadas para melhor permitir as alterações ao grafo?
- Quais as primitivas de grafos a usar na GPU para obter um desempenho eficiente?

1.3 Proposta de Solução

A biblioteca desenvolvida no contexto desta tese, o Marrow-Graph, oferece uma representação de um grafo que pode ser alterado em tempo de execução independentemente do estado atual do grafo. Para isso são disponibilizadas operações de adição, remoção e de edição de vértices e arestas de modo a ser possível manipular o grafo sem qualquer restrição. Também são disponibilizadas um conjunto de operações base, inspiradas no Gunrock, para poderem ser criados algoritmos de modo a ser possível analisar o grafo.

A implementação da biblioteca faz uso da biblioteca do Marrow [20] para a computação paralela em ambiente heterogéneos, de modo a produzir uma biblioteca de processamento de grafos dinâmicos em GPU. O Marrow é uma *framework* com a capacidade de tirar partido da GPU usando-a como um co-processador. Uma funcionalidade importante do Marrow é que gere de modo automático a consistência de dados entre CPU e GPU. Isto permite que a representação do grafo coexista na memória central do computador e na memória da GPU. Permitindo que o processamento do grafo pode ser realizado no *host* ou no GPU, cabendo ao Marrow esta decisão.

No entanto o Marrow não possui ferramentas desenhadas para o processamento de grafos, tendo que estas sejam estudadas e implementadas de modo a adicionar essas funcionalidades ao Marrow.

Também será implementado no Marrow uma estruturada de dados que permita representar um grafo, eliminando a necessidade de reconstrução total do grafo quando o mesmo sofre adições ou remoções de arestas ou vértices.

A avaliação experimental da nossa solução passa por uma comparação com outras bibliotecas de processamento de grafos que usem a GPU como um acelerador, comparando o desempenho desta solução dinâmica com outras disponíveis, como o Hornet, e também contra soluções estáticas, como o Gunrock.

1.4 Contribuições

As principais contribuições do trabalho realizado no contexto desta dissertação são:

- Oferecer uma biblioteca de processamento em GPUs que permite processar e aplicar algoritmos a grafos dinâmicos.
- Adição de novas funcionalidades à biblioteca Marrow. Estas funcionalidade são necessárias para a criação e processamento do grafo dinâmico, mas o seu uso não é restrito ao paradigma de processamento de grafos. Portanto estas funcionalidades foram adicionadas ao Marrow de forma a expandir as ferramentas que possui.
- Avaliação experimental da solução desenvolvida, comparando-a com outras bibliotecas de processamento de grafos em GPU.

1.5 Estrutura do Documento

Este documento está estruturado em 6 capítulos.

Neste primeiro capítulo introduzimos o documento e focamo-nos na motivação, explicando o problema e necessidade da sua respetiva solução. São estabelecidos contribuições e objetivos previstos.

No capítulo 2 fazemos um levantamento do estado da arte relevante para esta tese. É feita uma análise da arquitetura da GPU, um levantamento das diferentes estruturas de dados usadas para representar um grafo e um levantamento de os atuais modelos de programação em grafos. É aprofundado o processamento dinâmico de grafos em GPU, comparando o potencial dinâmico das estruturas de dados e analisando outra biblioteca de processamento de dados dinâmica. Por fim é feita uma análise em detalhe ao Marrow, discutindo a sua arquitetura e funcionalidades úteis para esta solução.

No capítulo 3 são discutidas duas possíveis representações para a estrutura de dados escolhida e como é que esta foi implementada no Marrow. É apresentado como é que o grafo processa as operações de adição, remoção e edição, de vértices e arestas. É também é explicado como foi feita a implementação das funções necessárias para processar um grafo no Marrow. O capítulo termina com uma breve conclusão dos pontos falados.

No capítulo 4 detalhamos as adições feitas ao Marrow que também podem ser usadas fora do contexto de grafos, o capítulo acaba com uma breve conclusão sobre estas adições.

No capítulo 5 avaliamos a nossa solução. São levantadas algumas questões e a sua reposta é encontrada através de alguns testes. É também definido a metodologia dos testes. É avaliado o resultado dos testes e as perguntas levantadas são respondidas na conclusão.

Por fim, no capítulo 6 apresentamos as conclusões finais do documento, onde é discutido se os objetivos foram cumpridos, por fim é proposto algum trabalho futuro.

2

Estado da Arte

Este capítulo foca-se numa análise às Unidades de Processamento Gráfico (GPUs), começando por uma comparação com o CPU, a nível arquitetónico sendo posteriormente analisados os principais componentes que constituem a GPU (secção 2.1.1). É, de seguida descrito o modelo de execução (secção 2.1.2), o modelo de programação (secção 2.1.3) e modelo base da GPU (secção 2.1.3.1). Por fim são referidas algumas otimizações e boas práticas de programação em GPU (secção 2.1.3.2).

Neste capítulo é ainda abordado algumas estruturas de dados comuns para a representação de grafos em GPU (secção 2.2), seguido de uma descrição e comparação de dois modelos de programação (secção 2.3). Num ponto posterior é feita a descrição do *cuSTINGER*, uma biblioteca de processamento de grafos dinâmicos e de como este suporta uma estrutura dinâmica em GPU (secção 2.4) sendo feito um levantamento de outras estruturas dinâmicas, também em GPU (secção 2.4.1).

Por fim a arquitetura do Marrow é apresentada (secção 2.5.1), tal como as Marrow *expressions* (secção 2.5.2), estruturas de dados (secção 2.5.3, operadores (secção 2.5.4), *skeletons* (secção 2.5.5) e por fim é exemplificado o seu modelo de execução (secção 2.5.6).

2.1 A GPU

2.1.1 Arquitetura da GPU

As GPUs são construídas especificamente para renderizar aplicações gráficas que têm um alto grau de paralelismo de dados. De uma forma geral, um núcleo de uma GPU tem uma frequência mais baixa e é mais simples. Por outro lado, um núcleo de um Unidade Central de Processamento (CPU) é mais complexo e tem uma frequência mais alta. Os núcleos do CPU são construídos com o objetivo de serem capazes de ter uma bom desempenho numa alta variedade de tarefas [16].

No entanto a simplicidade e frequências mais baixas são compensados com um maior número de núcleos disponíveis, pela parte da GPU. Isto é ideal para a computação de tarefas altamente paralelizáveis.

Figura 2.1: GPU Pascal GP100 completa, com 60 SMs

O CPU tem um desenho orientado a reduzir a latência, com unidades de lógica e aritmética poderosas, caches grandes e técnicas como o "*branch prediction* e "*data forwarding*". Por outro lado, a GPU tem um desenho orientado a rendimento, tem caches mais pequenas, maior número de unidades de lógica e aritmética com uma latência elevada, mas bastante *pipelined* para rendimento. Não tem técnicas como "*branch prediction*" e "*data forwarding*" [36].

Streaming Processor

Para ser possível tratar eficientemente de tarefas altamente paralelizáveis, uma GPU moderna consiste de vários SM, como ilustrado na figura 2.1.

Cada SM, como representado na figura 2.2, é composto por um determinado número de núcleos, unidades funcionais especiais Special Functional Units (SFU), registos, unidades de dupla precisão e um grupo de *threads*, denominado *warps*. Cada núcleo contém unidades para aritmética sobre inteiros e unidades de aritmética para vírgula flutuante onde a maioria das instruções da GPU são executadas [25].

Memória

A memória da GPU pode ser divida em dois níveis: memória no dispositivo e memória no *chip* (figura 2.3). A memória do dispositivo, também conhecida por memória de acesso aleatório dinâmica Dynamic Random-Access Memory (DRAM), encontra-se dividida em termos lógicos em memória local, memória constante e memória de texturas. O acesso à

Figura 2.2: Pascal GP100 SM

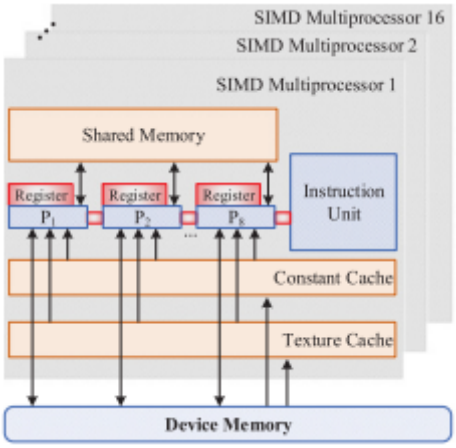

Figura 2.3: Arquitetura de memória na GPU [30]

memória no dispositivo habitualmente tem um acesso de longa latência, na ordem das centenas de ciclos de relógio.

A memória do *chip* contém componentes físicos diferentes, incluindo um registo de memória partilhada, cache *L1* e *L2*, memória constante e cache de texturas. Cada *thread* tem acesso a uma pequena e exclusiva parte de memória e todas as *threads* podem aceder à memória global. A memória partilhada em cada SM é de baixa capacidade, mas de alta velocidade e apenas disponível para *threads* daquele SM [30].

2.1.2 Modelo de Execução

O modelo de execução é dividido em duas partes:

- O ***host*** é responsável pela configuração e parametrização dos *kernels* e também é este que aloca e transfere a memória para o dispositivo (ou GPU).
- O ***kernel*** é um sub-programa ou uma rotina que executa no GPU.

Cada *kernel* é executado por uma grelha de *threads*. A grelha é divida em blocos e os blocos compostos por *threads*. Cada bloco é uma abstração de um conjunto de *threads* que são executadas em paralelo, as *threads* dentro do mesmo bloco correm no mesmo SM, e podem comunicar entre si usando a memória partilhada. A sincronização entre *threads* usando barreiras só é possível entre *threads* dentro do mesmo bloco.

A imagem 2.4 apenas mostra uma grelha em duas dimensões, mas é possível representar em 1, 2 ou 3 dimensões. A dimensão da grelha e dos blocos pode ser definida pelo utilizador ao invocar o *kernel*. Uma vez invocado, um bloco é atribuído a um SM, sendo que um SM pode executar vários blocos. Para mascarar a latência de algumas operações mais pesadas (por exemplo, ler da memória global) é possível executar operações de outros*warps* enquanto se espera pelo resultado do *warp* anterior, esta técnica é frequentemente chamada de *"latency hiding"* [26].

2.1.3 Modelo de Programação

As plataformas base de referência para a programação de GPUs são OpenCL e o CUDA.

OpenCL é uma plataforma para escrever programas paralelos para serem executados em diversos tipos de processadores, como CPUs, GPUs ou FPGAS, ou mesmo em *clusters* [18][4]. Por outro lado, CUDA é uma Application Programming Interface (API) que permite executar código em apenas *hardware* licenciado pela NVIDIA.

A terminologia difere entre CUDA e OpenCL, para simplificar nesta secção apenas é referida a nomenclatura usado para CUDA, mas a tabela 2.1 faz a correspondência entre as duas APIs:

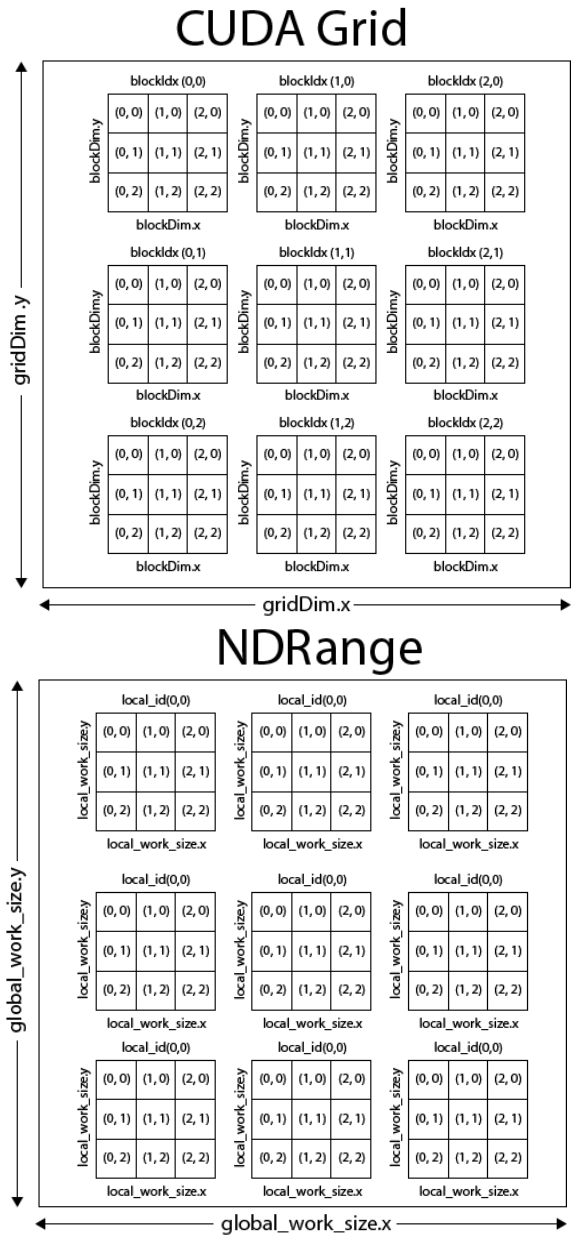

Figura 2.4: Uma grelha em CUDA e o NDRange em OpenCL [26]

Termo	CUDA	OpenCL
Índice do Bloco	blockIdx.[xyz]	get_group_id(uint)
Dimensão do Bloco	blockDim.[xyz]	get_local_size(uint)
Índice da Thread	threadIdx.[xyz]	get_local_id(uint)
Dimensão da Grelha	gridDim.[xyz]	get_num_groups(uint)
Memória Local	Memória Partilhada (shared)	Memória Local
Subgrupo de Threads	Block	Work-Group
Thread	Thread	Work-Item
Em que uint pode ser 0, 1 ou 2 correspondendo a x, y e z respetivamente.		

Tabela 2.1: Correspondência entre a terminologia utilizada nas APIs de CUDA e OpenCL [15]

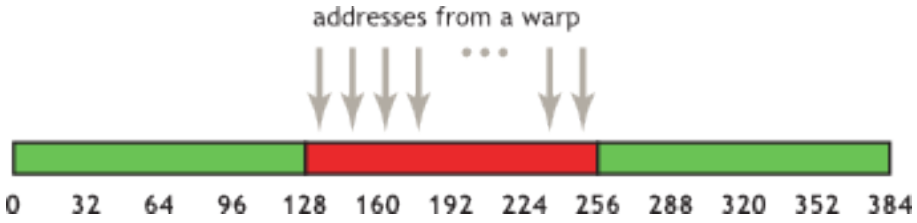

Figura 2.5: Acessos coalescidos [23]

2.1.3.1 Modelo Base

A memória da GPU é independente do resto da memória do sistema, por esta razão, o primeiro passo num programa habitual em GPU passa por alocar memória no CPU e GPU; de seguida, transferir os dados da memória do CPU para a memória do GPU a partir do *PCI-e Bus*, de seguida carregar e executar o programa na GPU; quando o programa finalizar a sua execução, copiar o resultado da memória do GPU para a memória do CPU e no fim libertar a memória no CPU e GPU.

Devido ao facto de cada núcleo ser relativamente fraco em termos de capacidade de processamento e uma GPU possuir um grande número de núcleos, torna-se desejável atribuir uma pequena quantidade de trabalho a cada núcleo. Sabendo os valores dos atributos na tabela 2.1, é possível identificar a posição de uma *thread* na grelha. Isto torna-se útil, por exemplo, na adição de dois vetores, em que é desejado que cada *thread* trate de apenas uma adição, sendo a posição do índice para cada *thread* calculado a partir da seguinte fórmula: `blockIdx.x * blockDim.x + threadIdx.x`

2.1.3.2 Otimizações e boas práticas de programação

Sobreposição de computação com comunicação – Ao sobrepor transferências de dados entre o *host* e o dispositivo, com computação no dispositivo e computação no *host* é possível obter acelerações significativas no tempo de execução. Esta otimização é tratada automaticamente pelo Marrow [13].

Memória global – Os acessos a memória global são lentos, por isso é importante ao aceder à memória global fazê-lo da forma mais eficiente possível. Ao ter acessos de memórias coalescidos é possível servir várias *threads* de uma vez, a figura 2.5 ilustra esse mesmo caso.

Memória local – Por estar no chip os acessos à memória local são uma ordem de magnitude mais rápida do que aceder à memória global. A memória local é alocada por bloco, o que permite a todas as *threads* (dentro de um bloco) acederem a essa memória. A memória local também pode ser utilizada com uma cache gerida pelo programador. Esta está organizada em 32 bancos de memória independentes, no entanto é preciso evitar sempre que possível fazer acessos concorrentes ao mesmo

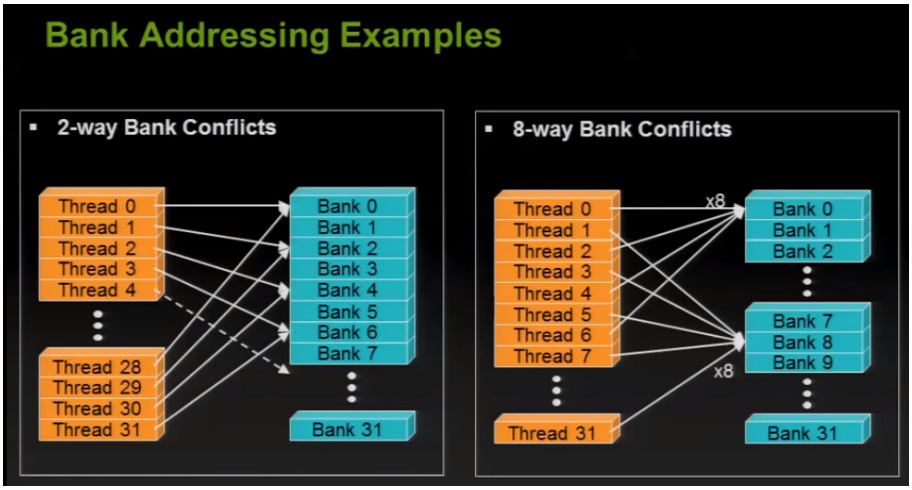

Figura 2.6: Conflito de bancos [22]

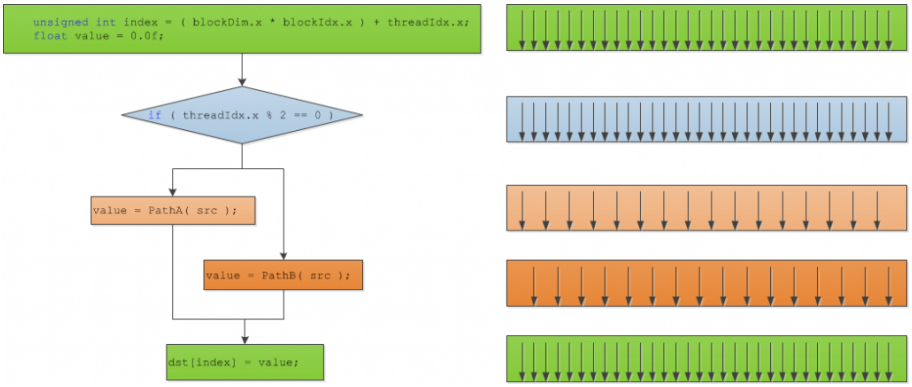

Figura 2.7: Divergência de *warps* [26]

banco; quando isto não é possível, os pedidos acabam por ser processados sequencialmente. A figura 2.6 esquematiza um conflito de bancos.

Divergência de warps – É natural achar que um *kernel* tem declarações para o controlo de fluxo, tal como *IF-THEN-ELSE*, *switch*, *while* e *for*. No entanto ao adicionar estas declarações introduzimos a possibilidade de criar divergências entre *warps*. A figura 2.7 esquematiza o que é uma divergência de *warps*, quando há um controlo de fluxo, todas as *threads* em que a condição é verdadeira são executadas e, posteriormente, todas em que a condição é falsa. Devido a esta execução sequencial, sempre que possível é crucial evitar as declarações de controlo de fluxo.

Sincronização entre blocos – Em CUDA apenas são suportadas sincronizações dentro do mesmo bloco. Para ser possível ter sincronização global entre *threads* em blocos diferentes, a computação pode ser divida entre *kernels*; visto que os *kernels* são executados sequencialmente na GPU, cada *kernel* atua como uma barreira global. Mas seja a barreira feita ao nível de bloco ou simulada globalmente com vários

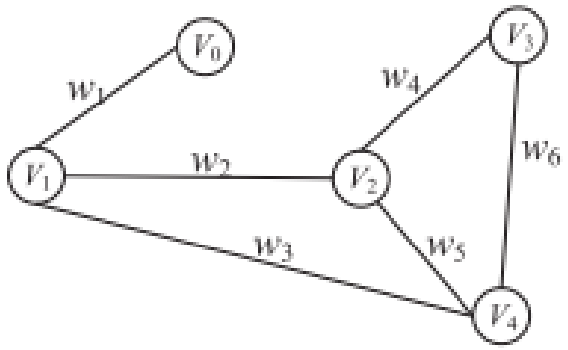

Figura 2.8: Grafo a ser considerado para os seguintes exemplos [30]

kernels, são operações caras, o que faz com que seja crucial desenhar algoritmos que tentem minimizar o número de pontos onde seja necessária uma sincronização.

2.2 Representação de Grafos em GPU

A obtenção de uma representação do grafo que seja simultaneamente compacta e que permita um acesso regular é um dos principais pontos a ter em conta no processamento destas estruturas de dados.

Uma vez que a memória disponível no GPU é tipicamente mais limitada do que no *host*, é crucial que a representação do grafo seja o mais compacta possível, com o bónus acrescido de que uma representação compacta traduzir-se-á num menor uso de largura de banda *PCi Express*. Adicionalmente, uma representação que permita acesso regular é crucial para tirar maior partido das capacidades altamente paralelizáveis da GPU.

Atualmente, as principais representações para algoritmos e sistemas de processamento de grafos em GPU são:

Matriz de adjacências [12, 33, 21, 3] – A partir de uma matriz quadrada, ao atribuir um valor diferente de 0 numa posição a_{ij} indicamos uma aresta do vértice v_i para o vértice v_j. Se o grafo for ponderado, o valor representa o peso entre os dois vértices. Para grafos grandes o resultado é tipicamente uma matriz esparsa, o que faz com que memória seja desperdiçada. No entanto simplifica alocação de memória para os programadores.

Lista de adjacências [12, 33, 21, 3] – Uma coleção de listas não ordenadas, em que se na posição 2 estiver uma lista {4,3} significa que há uma aresta do vértice v_2 para v_4 e v_2 para v_3. Esta abordagem é mais compacta que uma matriz de adjacências, no entanto não permite acesso regular, porque os vizinhos de diferentes vértices não estão ligados em espaço contíguo de memória. A figura 2.9 ilustra uma lista de adjacências com base na figura 2.8.

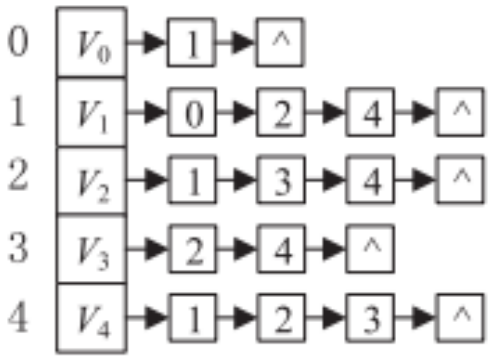

Figura 2.9: Lista de Adjacências [30]

Index	= [0	1	2	3	4	5	6	7	8	9	10	11]
Vertex	= [1	2			3			4		5]		
Segment-descriptor	= [1	3			3			2		3]		
Cross-pointers	= [1	0	4	9	2	7	10	5	11	3	6	8]
Weights	= [w_1	w_1	w_2	w_3	w_2	w_4	w_5	w_4	w_6	w_3	w_5	w_6]

Figura 2.10: V-Grafo [30]

V-Grafo [5] – A representação em v-grafo passa em guardar os dados num vetor segmentado, em que cada segmento corresponde um vértice. Para cada segmento o seu tamanho faz equivalência ao grau de cada vértice, e a partir dos apontadores cruzados é possível definir os dois vértices para cada aresta. No entanto cada vértice guarda a informação para cada aresta, porém origina informação repetida e reduz a sua compactidade. Como todas as arestas são guardadas num espaço contíguo de memória e ordenados pelos índices dos vértices, permite acessos regulares de memória.

Compact Sparse Row (CSR) [21] – Com 3 vetores, em que cada um guarda valores diferentes de 0. O vetor de "deslocamento de linha" guarda os índices válidos para cada vértice, ao subtrair $v_{i+1} - v_i$ ficamos com o grau do vértice i. O vetor "índice de colunas", guarda as arestas que saem do vértice. Por fim o vetor de valores guarda os pesos das arestas, caso o grafo seja ponderado. Como o *V-grafo* o CSR ordena e guarda a informação de todas as arestas compactamente num pedaço de memória contíguo, permitindo assim acesso de memória regular, baixa utilização de memória e reduz a largura de banda *PCi Express* quando a data é transferida entre o *host* e o dispositivo.

Slotted Pages [11] – Uma *slotted page* é tipicamente um espaço continuo de memória e está divido em duas secções: *records* e *slots*. A zona *records* é usada para guardar as listas de adjacências de cada vértice e as *slots* como mecanismo para identificar

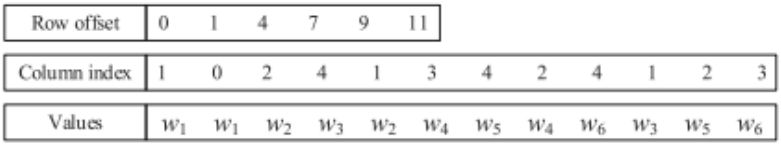

Figura 2.11: Compact Sparse Row (CSR) [30]

um vértice e para endereçar a respetiva lista de adjacências dentro da *slotted page*. Cada secção cresce num sentido oposto da outra, as *slots* estão tipicamente no fim da *slotted page* e crescem em direção ao seu início, os *records* por sua vez são colocados no início e crescem em direção ao fim da *slotted page*.

Uma *slotted page* é de tamanho fixo e é possível que na adição de vértices e arestas ao grafo esta fique cheia, para permitir o crescimento do grafo *slotted pages* adicionais são criadas. No entanto, e dado o seu tamanho ser fixo, é possível que uma página não seja suficientemente grande para guardar todos os vértices de uma lista de adjacências de um determinado vértice. Nesse caso são criadas páginas extra para armazenar todos os vértices adjacentes, estas páginas usadas para guardar extensas listas de vértices são apelidadas de *Large Adjacency Pages* (LA Pages).

Na concessão de estruturas de dados para grafos estáticos um ponto importante é a sua compactidade, este ponto desce de consideração quando o paradigma é alterado para grafos dinâmicos mesmo que o espaço disponível na GPU seja tipicamente menor do que na CPU e por sua vez mais importante gerir eficientemente.

Uma estrutura compacta por definição não tem espaço para permitir que o grafo cresça, o que é importante para evitar a reconstrução íntegra da estrutura de dados e consecutivamente a sua de retransmissão para o GPU, visto que esta operação tem um computacionalmente elevado. Outra preocupação é o comportamento destas estruturas de dados quando inevitavelmente ficam cheias, se as estruturas alocam um espaço contínuo de memória é necessário ao crescerem alocarem um espaço maior, copiar o conteúdo da estrutura de dados antiga para a nova e libertar o espaço anterior.

Com base nestes pontos podemos descartar o *V-grafo* e o CSR, pois qualquer adição no grafo obriga à reconstrução da estrutura.

A Matriz de adjacências permite a remoção ou edição de arestas e a remoção de vértices sem necessidade de alocar mais memória, no entanto, para adicionar vértices é necessário alocar mais memória, que se traduz numa reconstrução e retransmissão da estrutura para garantir que esta fique em memória contínua.

A lista de adjacências cria um novo espaço de memória para cada adição de um vértice ou aresta sem necessidade de copiar informação antiga para o novo espaço, porém não permite acesso direto aos identificadores dos vértices dentro das listas, criando acessos não regulares.

O principal inconveniente do *array* de adjacências é quando na inserção de arestas o *array* fica sem espaço disponível para guardar mais um vértice, sendo necessário a criação de um *array* com maior dimensão, seguido de uma transferência do conteúdo presente no *array* antigo para o novo *array* antes da sua libertação em memória. Mesmo com este inconveniente é sem dúvida uma estrutura de dados que se aplica ao modelo de grafos dinâmicos em GPU.

As *slotted pages* ao não serem compactas permitem que o seu espaço livre seja gradualmente utilizado sem que seja necessária a alocação de mais espaço imediato, quando é estritamente necessário criar uma nova página no caso de uma adição de vértices, não é necessário nenhuma cópia para que o estado do grafo seja mantido, no entanto na adição de arestas é normal haver uma transferência de informação de uma *slotted pages* para outra sem a remoção da página anterior.

As *slotted pages* foram escolhidas para a implementação de grafos dinâmicos no Marrow, e a sua implementação explicada na secção 3.2, uma vez que através do *design* têm embutido espaço para alterações ao grafo. Cada página é um espaço contínuo de memória que permite acessos regulares e devido à sua capacidade de se adaptar a uma grande configuração de grafos, uma vez que o tamanho das *slots* e dos *records* é ajustado automaticamente e dinamicamente em cada página, à medida que o grafo cresce.

2.3 Modelos de Programação em Grafos

2.3.1 Gather Apply Scatter

Neste modelo de programação as operações são feitas no vértice.

A fase *Gather* – Nesta altura o vértice recolhe informação dos vértices vizinhos através de uma função definida pelo programador.

A fase *Apply* – Uma função definida pelo utilizador é aplicada a todos os vértices com base na informação recolhida na fase de *gather*. Esta parte não tem qualquer comunicação com outros vértices.

A fase *Scatter* – Os novos valores dos vértices são espalhados (*scattered*) para os vértices e arestas adjacentes. Em algumas implementações, a atualização do novo valor não é restringida aos vértices e arestas adjacentes, sendo permitida a atualização a vértices e arestas remotos. Em alguns algoritmos a atualização remota faz com que a fase de *gather* não seja necessária.

O modelo Gather Apply Scatter (GAS) abstrai a sobrecarga na sincronização, o que simplifica o processo de análise e a exatidão do algoritmo ao usar este modelo. Como referido na secção 2.1.3.2, a sincronização entre blocos é cara e não deve ser ignorada.

Este modelo é usado em vários sistemas de processamento de grafos tais como: VertexAPI2, MapGraph, CuSha e GraphReduce [10]. Estes sistemas tipicamente disponibilizam uma API com um modelo centrado no vértice.

2.3.2 Bulk Synchronous Parallel

Executado numa sequência de fases chamadas *super-steps*.

Consiste em 3 fases:

Computação local – em que as tarefas são executadas localmente.

Comunicação global – todas as mensagens emitidas e recebidas são executadas nesta fase.

Fase de sincronização – garante que as fases anteriores estão completas e sincronizadas nesta fase.

Neste modelo, em cada super passo, uma função definida pelo programador é aplicada a cada vértice de maneira assíncrona, isto faz com que o valor do vértice mude e a sua atualização é passada aos vizinhos. Habitualmente ao usar este modelo o grafo é dividido em várias secções e um *kernel* definido pelo utilizador vai ser executado para cada secção. Num super passo todas as *threads* correm concorrentemente. Dentro do *kernel*, cada *thread* recebe mensagens do último super passo e posteriormente efetua a computação local.

Na fase da comunicação, o valor dos vértices é guardado em memória local para minimizar transferências de dados. Cada *kernel* pode, se necessário, enviar mensagens para os seus vizinhos antes do final de cada super passo. Uma barreira é imposta entre os dois super passos para sincronizar todos os *kernels*. A principal desvantagem a usar o modelo Bulk Synchronous Parallel (BSP) é que a *thread* que demora mais tempo a executar vai atrasar todas as outras *threads* naquele super passo.

Este modelo foi primeiramente adotado pelo sistema Pregel [19] (com uma abordagem centrada no vértice) e posteriormente o modelo foi incluído em TOTEM [8], Medusa [37] e Gunrock [34]. O Gunrock enumera também um conjunto de operações base para a execução de algoritmos todas estas operações seguem o modelo BSP.

Advance – Gera uma nova lista de vértices ao expandir todos os vértices colocados como *input* e devido à natureza dos grafos o tamanho do *output* não é fixo.

Filter – Filtra a fronteira com base numa condição definida pelo programador, cada valor na fronteira de *input* pode originar um ou zero valores na fronteira de *output*.

Segmented Intersection – Recebe duas fronteiras com o mesmo tamanho, e para cada par de vértices é computado as interseções (os vértices em comum) das suas expansões.

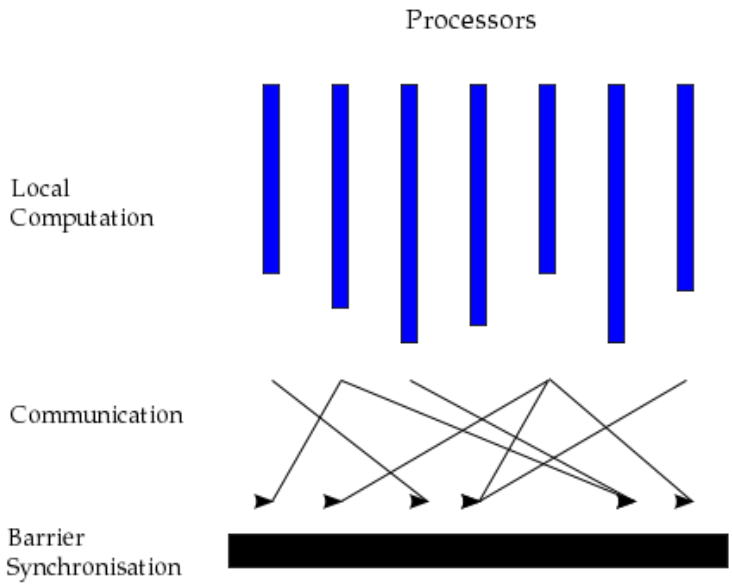

Figura 2.12: Modelo BSP [35]

Compute – Aplica uma função definida pelo utilizador a todos os elementos da fronteira, o tamanho do *output* é igual ao tamanho do *input*.

Estas operações foram adcionadas ao Marrow sobre a forma de *skeletons* e a sua implentação é detalhada na secção 3.4. A sua implementação representa um acréscimo de funcionalidades à biblioteca Marrow.

2.4 Processamento Dinâmico de Grafos no GPU

Uma abordagem para a obtenção de grafos dinâmicos pode passar pelo reprocessamento de um grafo estático após uma atualização, no entanto esta abordagem é computacionalmente cara e não é viável para grafos grandes.

O foco dos grafos dinâmicos é analisar grafos em constante mudança, seja por adições, remoções ou atualizações, desenhar um algoritmo capaz de evitar a re-computação total ao usar um estado anterior é desejado, mas nem sempre possível. No entanto quando possível, algoritmos para grafos dinâmicos são consideravelmente mais rápidos que os seus homólogos estáticos.

Para o suporte de algoritmos dinâmicos é tipicamente necessária a existência de estruturas de dados dinâmicas. Estas estruturas de dados necessitam de um balanço na quantidade de itens a serem adicionados. A adição de um único elemento (ou um baixo número), apesar de possível, é um desperdício dos recursos disponibilizados pela GPU, no outro extremo do espectro uma adição de um grande número de elementos pode criar

Algorithm 1: Pseudo code for updating a cuSTINGER graph with a set of B edge insertions. Functions followed by <<<>>> denote a call to a GPU kernel.

Inputs: $cuSting$ - dynamic graph.
Inputs: B - batch of updates.
$CopyHostToDevice(B)$
$Phase1Update$ <<<>>> $(cuSting, B, |B|)$// GPU kernel

$CopyDeviceToHost(\hat{B})$ // Uncompleted edge list
$CopyDeviceToHost(Dups_B)$ // Duplicated inserted edges in B
$RemoveDuplicates$ <<<>>> $(Dups_B)$ // GPU kernel

// Reallocates memory for all unique sources vertices of $\hat{B}$.
$CopyDeviceToHost(PointerAdjacencyList)$
CPU-ReallocateMemory($\hat{B}$, $PointerAdjacencyList$)
$CopyHostToDevice(NewPointerAdjacencyList)$

// Copies entire edge lists for all vertices new edge lists
$DeepCopyDeviceToDevice(PointerEdgeList, NewPointer-EdgeList, \hat{B})$
$Phase2Update$ <<<>>> $(cuSting, \hat{B}, \hat{B})$// GPU kernel

Figura 2.13: Pseudo-código para adição de arestas no cuSTINGER [9]

um custo elevado na reestruturação de dados que não compensa fazer a adição do lado da GPU, mas sim no *host* e depois transmitir os dados de volta para a GPU. Uma adição com um número moderado de elementos é o ideal pois permite tirar partido das capacidades paralelas da GPU e sem ser necessário uma reestruturação de elevado custo.

No caso do cuSTINGER [9], uma biblioteca de processamento de grafos dinâmicos, usa uma abordagem que tira o melhor da matriz de adjacências e da lista de adjacências. CuSTINGER usa um *array* de adjacências para cada vértice, o que faz com que a memória seja reservada num espaço continuo tendo uma melhor utilização de memória comparativamente a uma matriz de adjacências.

As remoções de arestas no cuSTINGER são relativamente diretas, uma remoção nunca irá necessitar de alocar mais memória. A remoção é feita em duas fases: numa primeira fase são localizadas e marcadas como apagadas, na segunda fase é copiado o último elemento de cada lista para a posição da aresta apagada.

Nas adições, primeiro é verificado se a aresta já não se encontra no grafo e se houver espaço, todas as arestas não duplicadas são inseridas no grafo. No entanto para cada vértice que não tenha espaço no seu *array* recebe um novo e maior *array*. Isto é feito a partir de uma série de cópias de memória entre o dispositivo e o *host*. Após a alocação de memória, é efetuada uma cópia do *array* antigo para o novo e, no fim, a inserção das novas arestas. A imagem 2.13 apresenta o pseudo- código usando a biblioteca cuSTINGER na inserção de um lote de elementos.

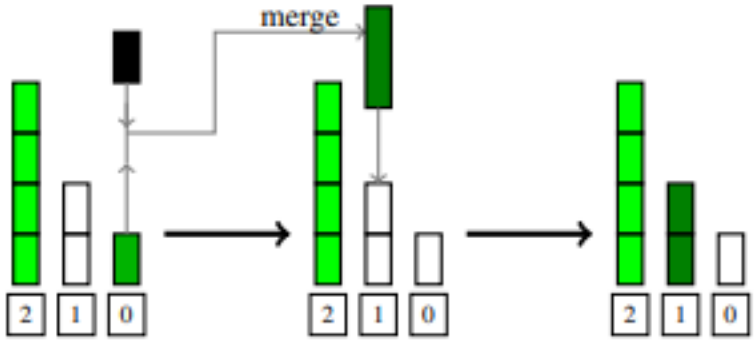

Figura 2.14: Adição de um lote em dicionário dinâmico em GPU [2]

2.4.1 Outras Estruturas Dinâmicas em GPU

Dicionário dinâmico em GPU Formalmente, um dicionário mantém um conjunto de pares $\{chave, valor\}$ em que as chaves estão ordenadas. O trabalho apresentado por Ashkiani *et al* em [2] propõe um dicionário desenhado com o propósito de ter adições eficientes em GPU, para tal, como anteriormente discutido o número de atualizações tem de ser moderado. As atualizações são aglomeradas até terem o tamanho b necessário para criar um lote. A estrutura em si é definida por um conjunto de *arrays* ordenados em diferentes níveis, tendo cada nível o dobro do tamanho do anterior, o nível mais pequeno consiste num *array* ordenado de tamanho b.

As adições deste modo são simples: se o nível mais baixo estiver vazio o lote é adicionado aí, caso contrário o nível anterior e o lote são ordenados e há uma tentativa de o colocar no nível posterior, e o ciclo repete-se. No caso de o último nível se encontrar cheio é possível alocar mais um nível sem necessitar que os dados sejam retransmitidos para o *host*.

As remoções são feitas ao inserir um elemento especial do tipo lápide, o que faz com que as remoções sejam efetivamente, tratadas como uma adição. Para remover uma chave e o seu respetivo valor é apenas necessário adicionar o elemento lápide com a mesma chave que o valor que pretendemos apagar.

Esta estrutura certifica-se que as chaves mais recentes são encontradas primeiro que as mais antigas. Por isso a procura é sempre iniciada nos níveis mais baixos e propagada até ser encontrada.

Estruturas esparsas dinâmicas Estruturas esparsas são construídas ao dividir volumes tridimensionais em pequenos segmentos bidimensionais. Para estas estruturas é proposto que o CPU armazene estes segmentos na memória da GPU [24]. A GPU que processa os segmentos e o CPU é responsável pela alocação e libertação de memória, desta forma conseguimos obter atualizações dinâmicas ao usar o CPU como gestor de memória para

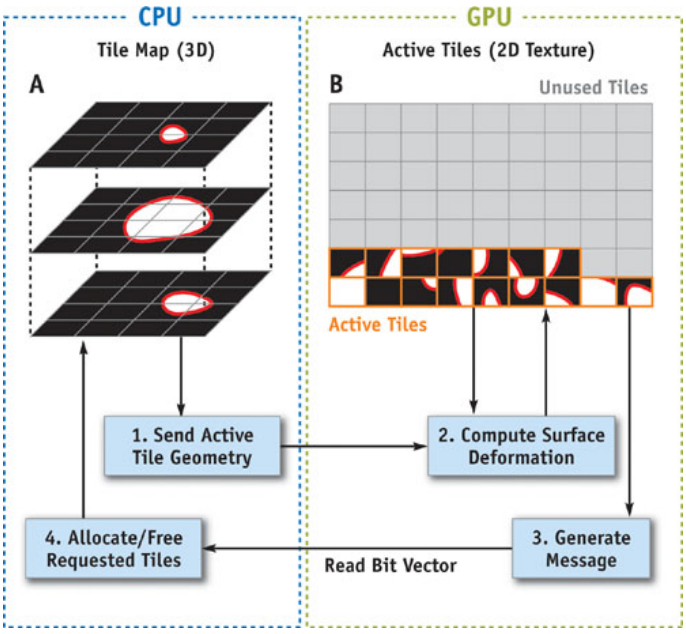

Figura 2.15: Dados de estrutura esparsa dinâmica em GPU [24]

a GPU. Uma componente chave deste sistema é o facto da alocação ou libertação de segmentos ser requisitada à CPU pela GPU. O requerimento é feito a partir de uma mensagem comprimida de modo a minimizar a transferência entre GPU e CPU.

2.5 Marrow

O Marrow [20, 1, 32] é uma *framework* de *skeletons* para o C++, com a capacidade de tirar partido de coprocessadores, como a GPU, de modo a alcançar programação heterogênea. O Marrow permite a aglomeração de *skeletons* que, por sua vez, permite ao programador alcançar uma camada de abstração da complexidade das computações a serem executadas. Nesta secção, a arquitetura, modelo de execução e os *skeletons* do Marrow vão ser apresentados e contextualizados para o objetivo de criar um grafo dinâmico em GPU.

2.5.1 Arquitetura

A arquitetura do Marrow, ilustrada na figura 2.16, está dividida em quatro camadas principais: Marrow, "Expressions", "Runtime" e "Backend". Sendo que uma aplicação construída usando a *framework* do Marrow só tem comunicação direta com as camadas mais superiores (Marrow e "Expressions"). Estas duas camadas comunicam com o "Runtime" com a passagem de *kernels*, o "Runtime" é então responsável pela verificação de erros do código OpenCL e se possível, da fusão e posterior compilação dos *kernels* para código binário. Finalmente a atribuição do código aos dispositivos compatíveis com OpenCL é feito no "Backend".

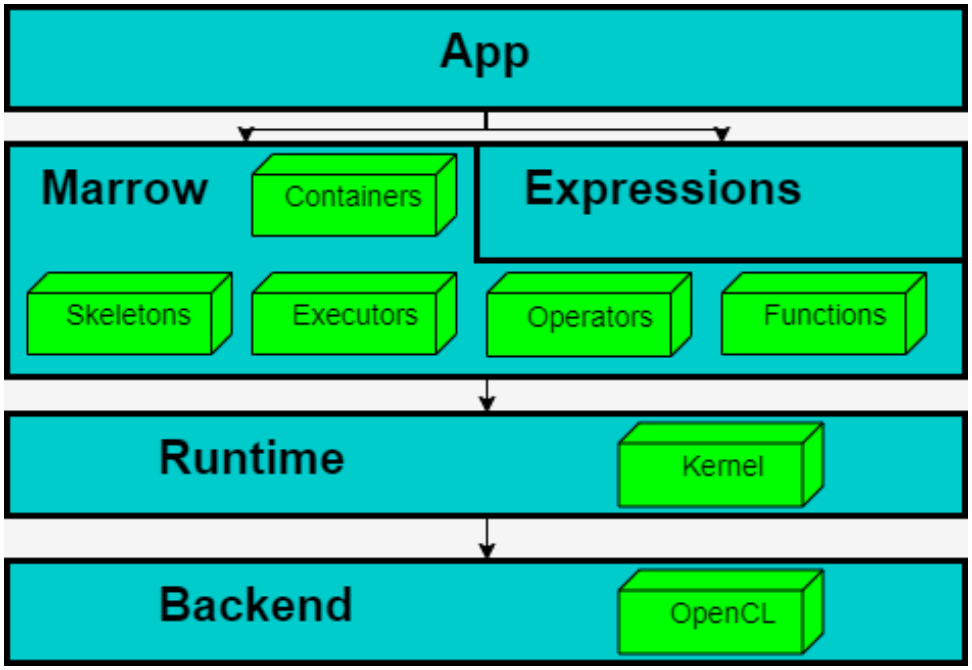

Figura 2.16: Diagrama da arquitetura do Marrow

Listing 1 Exemplo da criação de uma AST no Marrow

```
1 using namespace marrow;
2
3 marrow::array<int, N> a;
4 auto b = a * 2;
5 auto c = a + b;
6 marrow::scalar <int> x = reduce<max>(c);
7 (...)
8 cout << x.value() << endl;
```

2.5.2 Expressions

A Marrow expressions é uma camada acedida diretamente pelo programador, é a base de construção de todas as operações, funções e *skeletons*. Todas estas operações são desenhadas para correrem tanto em GPU como em CPU, sendo possível compor várias operações diferentes sobre as estruturas de dados de uma forma automática ao se construir uma Abstract Syntax Tree (AST). Esta AST é responsável pela a ordem de execução das operações submetidas, assim como permitir a fusão de algumas operações, que pode trazer ganhos de desempenho significativos. A construção da AST é efetuado com o auxilio do operador auto do C++, que automaticamente define o tipo das expressões. A listagem 1 exemplifica a criação de uma AST.

Quando a AST é atribuída a uma estrutura de dados é quando desencadeia a geração, compilação e execução do código OpenCL, esta geração é gerida automaticamente pelo

Marrow, sem que o programador tenha que invocar diretamente o *kernel*. Desta forma colocando uma camada de abstração ao programador permitindo-o focar-se na implementação e ao mesmo tempo abstrair-se de todos os detalhes necessários para que seja possível executar código no GPU.

2.5.3 Estruturas de dados do Marrow

O Marrow possui as típicas estruturas para colecionar elementos, mas com o benefício acrescentado dessas mesmas estruturas de dados estarem presentes tanto na memória do dispositivo com na memoria do *host*. A sua persistência e atualização é gerida automaticamente sendo que uma série de operações em GPU não força a transferência de dados para o CPU, diminuindo eventuais sobrecargas da constante transferência de dados. Estas estruturas de dados replicam o comportamento da biblioteca *standard* do C++.

Array – Coleção de elementos de tamanho estático, em que o seu tamanho é conhecido a tempo de compilação.

Vector – Coleção de elementos de tamanho dinâmico, em que o seu tamanho é conhecido a tempo de execução.

Matrix – Coleção de elementos com várias dimensões, o seu tamanho pode ser estático ou dinâmico.

Scalar – Guarda apenas um valor, usado tipicamente para guardar uma resposta a uma computação de redução.

2.5.4 Operadores

O Marrow disponibiliza operadores aritméticos e relacionais, mas aplicados a estruturas de dados, é possível, por exemplo somar um vetor a outro, ou comparar os valores de uma estrutura de dados a uma constante. Estes tipos de operações são embaraçosamente paralelos obtendo *speedup* considerável quando executados na GPU.

2.5.5 Skeletons

Os *skeletons* suportados pelo Marrow são os seguintes:

Map – Aplica uma função, definida pelo programador, a todos os elementos de uma coleção. Esta função permite que cada valor seja calculado independentemente de outros.

Reduce – Também conhecido como MapReduce, é semelhante ao "Map" no sentido que uma função é aplicada, mas o resultado do mapeamento é tipicamente reduzido a um elemento. Por exemplo a soma de todos os valores num *array*.

Loop – O "Loop" aplica a mesma função a uma coleção de dados, num determinado número de vezes, semelhante a um ciclo *for*. O programador tem de definir um inteiro com o valor do início, outro com o valor final, e finalmente tem que definir um *step*. O *step* é incrementado no final de todos os ciclos ao valor inicial até que este chegue ao valor final pré-definido.

2.5.6 Modelo de execução

Uma das principais preocupações ao desenhar um sistema heterogéneo é o uso eficiente dos seus recursos, para se extrair o máximo de desempenho é crucial que o tempo de bloqueio seja o mais curto possível. A execução de *skeletons* é dissociada do programa principal, permitindo que este continue com a sua execução só bloqueando quando o resultado de uma computação é necessário para o progresso do programa principal. O Marrow respeita esta preocupação só desencadeando a execução numa atribuição a uma estrutura de dados, bloqueando apenas quando o valor é explicitamente acedido. No código exemplo da listagem 1, é dado um exemplo prático de como isto é executado.

Na linha 6 a execução do *kernel* é lançado assincronamente.

Na linha 7 podem estar ou não outros blocos de código. Se estiverem presentes são executados normalmente enquanto o *kernel* é processado.

Na linha 8, a execução é bloqueada até que o resultado tenha sido computado.

3

Grafos Dinâmicos no Marrow

Neste capítulo é apresentada a nossa solução para o suporte de grafos dinâmicos em GPU, tendo por base a plataforma Marrow. Começamos por apresentar duas representações para um grafo em GPU (secção 3.1).

Também é apresentada a implementação do grafo no Marrow (secção 3.2) e as funções necessárias para tornar o grafo dinâmico (secção 3.3). No fim é explicado como foram implementadas as funções base necessárias para processar o grafo (secção 3.4).

3.1 Representação de um Grafo Dinâmico em GPU

Como referido na secção 2.2, iremos utilizar as *slotted pages* como a base para a representação de grafos dinâmicos em GPU. Sendo nesta secção discutidas duas representações consideradas para uma *slotted page*. A primeira representação foca a minimização do espaço possível para a representação de um grafo. A segunda representação, por sua vez, sacrifica alguma eficiência espacial em troca de uma representação que permite aceder eficientemente ao início de cada lista de adjacências. A primeira representação é referida como ME (Minimização de espaço) e a segunda representação como AE (Acesso eficiente).

Neste contexto a ME guarda apenas as listas de adjacências na zona das *slots*, sem espaços livres entre diferentes listas de adjacências. Na zona das *records* para cada vértice são guardados dois valores: o identificador do vértice e o seu grau.

Por seu lado, a representação AE, para além da zona das *records* e da zona das *slots* esta representação usa as primeiras 3 posições da *slotted page* para o cabeçalho. Este cabeçalho guarda na primeira posição guarda o número de vértices na página, o índice do último valor da zona das *slots* na segunda posição e na terceira posição está presente um "1" se a página fizer parte de uma grande lista de adjacências, ou "0" no caso contrário. Na zona das *slots* é guardado o grau do vértice seguido da lista de adjacências desse mesmo vértice. Na zona das *records* também são guardados dois valores para cada vértice: o identificador do vértice e um valor que representa o índice onde está o grau deste vértice (relembrando que a lista de adjacências está imediatamente a seguir).

As representações ME e AE do grafo ilustrado na figura 3.1 são apresentadas nas

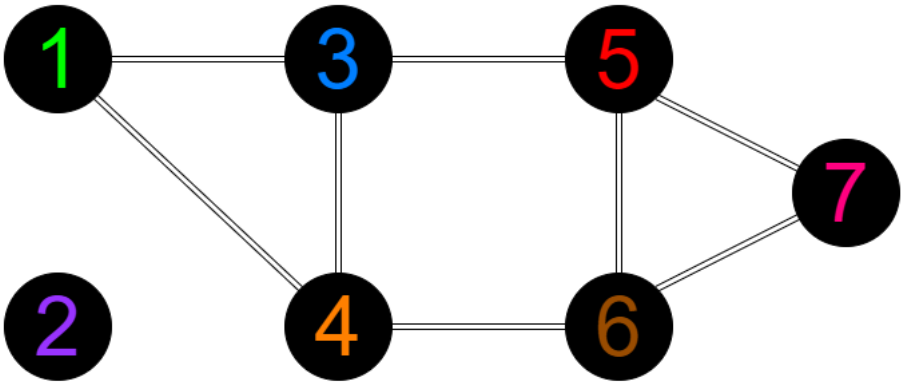

Figura 3.1: Grafo a ser considerado

3	4	1	4	5	1	3	6
3	6	7	4	5	7	5	6
		D2	7	D3	6	D3	5
D3	4	D3	3	D0	2	D2	1

Tabela 3.1: Representação ME do grafo da figura 3.1

tabelas 3.1 e 3.2, respectivamente. As duas tabelas têm a zona das *records* a azul-claro e a zona das *slots* representada a verde-claro. Apenas a representação AE tem um cabeçalho, representado a laranja-claro. O espaço livre está representado a branco e neste exemplo apenas a representação ME tem espaço livre. Os graus dos vértices estão representados com um "D" seguido do seu valor.

Na representação ME, para calcular o índice onde começa a lista de adjacências para um vértice é necessário somar os graus dos vértices anteriores. Por exemplo, para descobrir-se em que índice começa a lista de adjacências do vértice 3 temos de somar os graus dos vértices 1 e 2 da *slotted page*, ou seja $2 + 0$. Com isto obtemos a informação de que o início da lista de adjacências para o vértice 3 começa no índice 2. O grau do vértice somado ao índice do início da lista de adjacências informa qual o índice do último vértice da lista de adjacências.

Na representação AE, o cálculo dos limites de uma lista de adjacências de um vértice é mais direto. Para calcular o início da lista de adjacências dum vértice apenas é necessário

7	25	0	D2	3	4	D0	D3
1	4	5	D3	1	3	6	D3
3	6	7	D3	4	5	7	D2
5	6	23	7	19	6	15	5
11	4	7	3	6	2	3	1

Tabela 3.2: Representação AE do grafo da figura 3.1

calcular em que índice está o grau do vértice. Esta informação está explicitamente guardada na página, diretamente ao lado do identificador desse vértice, sendo que o início da lista de adjacências está na posição imediatamente a seguir. Tal como na primeira representação, quando, ao início da lista de adjacências, é somado o grau do vértice obtém-se o índice do último vértice da lista de adjacências.

A capacidade da 1ª representação necessitar de menos memória vem de não usar o cabeçalho e do início das listas de adjacências ter de ser calculado. O cabeçalho não guarda informação do grafo, mas sim informação da página, por isso a sua inclusão não é obrigatória. Dado que a granularidade da alocação de memória é a *slotted pages*, é importante é salientar que as duas representações neste exemplo, ocupam exatamente a mesma memória, uma *slotted page* com 40 entradas. No entanto a solução ME possui espaço livre que permite receber mais adições, seja de vértices ou arestas, antes de ficar sem espaço e ter que ser alocada uma nova página. A alocação de outras páginas é inevitável à medida que o grafo cresce e idealmente é desejado uma utilização total da memória alocada, no entanto é preciso um balanço entre os ganhos de desempenho ao deixar algum espaço inutilizado propositadamente no presente para permitir o grafo crescer no futuro com a perda de eficiência espacial. Esta eficiência espacial, que pode ser calculada com uma divisão entre o espaço utilizado com o espaço reservado, não é fixa. À medida que o grafo vai sofrendo adições ou remoções, a eficiência espacial também se altera.

Se o objetivo for apenas representar um grafo em *slotted pages* usando o mínimo de memória possível a primeira representação é superior.

No entanto é preciso ter em conta as aplicações finais do grafo. Uma operação frequente na maioria de algoritmos aplicados ao grafo é a obtenção da lista de adjacências de um ou vários vértices.

Nesse contexto, a segunda representação consegue obter essa informação mais rapidamente, pois não necessita de nenhum passo adicional para calcular o início da lista de adjacências de um vértice, visto que essa informação é explicitamente escrita nas páginas, ao contrário da maneira implícita da primeira representação. O cabeçalho também tem

como objetivo auxiliar a aplicação de diferentes funções ao grafo. A informação que está no cabeçalho não é sempre usada, mas quando é necessária, já está contida na *slotted page* ao contrário da primeira representação que tem de recolher e agregar essa informação para ser possível aplicar certas funções, precisando de mais tempo comparativamente à segunda solução.

Pesados as vantagens e desvantagens de ambas representações, optámos por implementar a AE no Marrow, por encontrar um bom equilíbrio entre eficiência espacial e desempenho comparativamente à primeira solução. Apesar da primeira solução conseguir representar grafos de uma maneira mais compacta do que a segunda, a sobrecarga para encontrar o início das listas de adjacências é demasiado significativa para ser ignorada. Ao referenciar diretamente o índice do início da lista de adjacências, a segunda representação consegue eliminar esta sobrecarga usando apenas mais uma posição da *slotted page* por vértice. O mesmo pode ser obtido com o cabeçalho, apesar da sua utilização não ser tão frequente como o calculo do início da lista de adjacências, são valores usados o suficiente para valer a pena embuti-los na *slotted page* para tornar o acesso mais direto e rápido. A memória na GPU é tipicamente reduzida em comparação à memória do resto do sistema e o seu aproveitamento é sem dúvida importante. No entanto é preciso ter em mente que a sua utilização tem como finalidade diminuir o tempo de execução do código. Sacrificar um pouco de eficiência espacial a favor de uma diminuição do tempo de execução é compatível com os objetivos desta tese, tornando a representação AE a escolha mais indicada.

3.2 Implementação do Grafo

A estrutura de dados do Marrow escolhida para implementar a *slotted page* foi o `marrow::array`, visto que o tamanho de uma *slotted page* é fixo e definido a tempo de compilação. O Marrow possui otimizações que permitem tirar partido do facto dos tamanhos dos dados serem conhecidos em tempo de compilação, aumentando o desempenho da aplicação. Esta abordagem impossibilita o redimensionamento de uma *slotted page* em tempo de execução e faz com que seja necessário ponderar o tamanho a escolher. A escolha de um valor demasiado grande resultar no desperdício de memória. Por outro lado, a escolha de um valor demasiado pequeno irá fazer com que a criação de novas páginas seja mais frequente, aumentando o tempo de execução das operações de manipulação do grafo.

A utilização de um *array* do Marrow permite que a informação possa ser acedida e manipulada tanto em GPU e CPU, sendo o Marrow responsável pela sincronização dos dados. Com isto é possível definir o grafo como uma coleção de páginas, sendo o grafo implementado definido como um vetor da biblioteca standard do C++. A utilização de um vetor permite que o número de páginas seja dinâmico durante a execução do programa sendo possível adicionar novas páginas ao grafo usando a função `emplace(pos)` que aloca

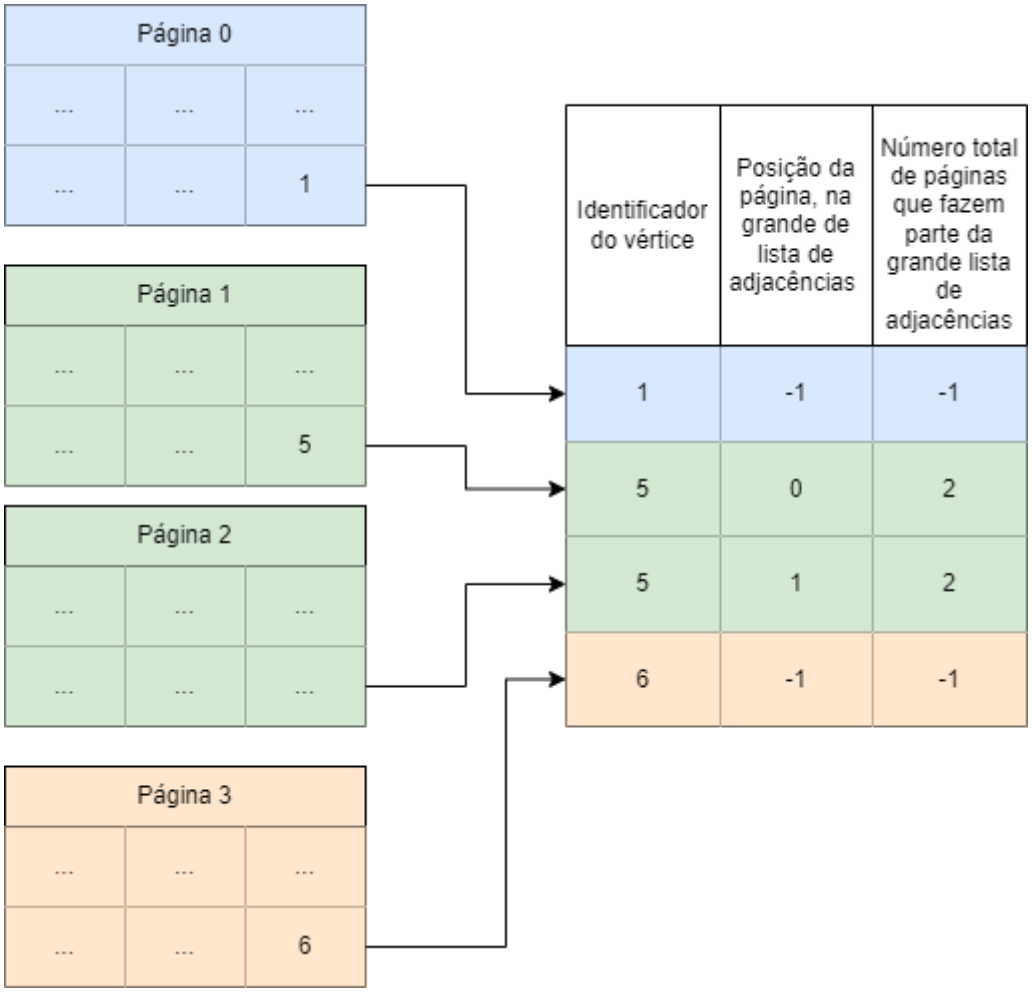

Figura 3.2: Representação do índice e páginas

uma nova página na posição pos e da função `erase(pos)` que apaga a página na posição pos, libertando a sua memória e reduzindo o tamanho do vetor.

Para auxiliar a localizar em que páginas se encontram os diferentes vértices no grafo foi implementada uma estrutura auxiliar, denominada de índice, semelhante ao que é proposto no TurboGraph [11]. Para cada página há uma entrada no índice, sendo esta entrada composta por três valores. O primeiro valor é o primeiro vértice presente na respetiva página. Os restantes dois só são necessários caso a página faça parte de uma grande lista de adjacências e guardam: a posição de página na grande lista de adjacências e o número total de páginas que fazem parte dessa lista. Caso a página seja uma *slotted page* isolada, o valor de ambas estas posições é "-1".

A figura 3.2 ilustra a implementação do índice. As duas páginas representadas a verde-claro fazem parte de uma grande lista de adjacências e as duas restantes são páginas isoladas que têm mais do que um vértice na sua zona das *records*. Para se efetuar uma busca no índice, tira-se partido do facto do índice estar em ordem crescente, sendo feita uma adaptação da busca binária. Em vez de se procurar um valor específico, procura-se

Listing 2 Definição do grafo em C++ usando a *framework* do Marrow

```
1  /**
2   * Slotted Page representation
3   *
4   * @tparam Size - size of the slotted page
5   */
6  template<size_type Size = default_slotted_page_size>
7  class slotted_page : public marrow::array<int, Size>;
8
9  /**
10  * Graph representation
11  *
12  * @tparam SlottedPageSize - size of the slotted page
13  */
14 template<size_type SlottedPageSize = default_slotted_page_size>
15 class graph : public std::vector<slotted_page<SlottedPageSize>>;
16
17 /**
18  * Index representation
19  */
20 using index = marrow::vector<int>;
```

se um valor é maior ou igual do que a página atual e simultaneamente menor do que a página anterior. A posição no índice reflete a posição da página no grafo.

Para a implementação do índice foi usado o `vector` do Marrow, pois o índice é dinâmico e necessita de estar presente tanto na memória do CPU como na da GPU.

Um grafo é portanto definido como uma coleção de *slotted pages* mais a estrutura auxiliar do índice. A listagem 2 mostra como é que em código C++ é usada a *framework* do Marrow para definir estas estruturas. Nas linhas 6 e 7 está a definição da *slotted page*, onde na linha 6 está definido o tamanho da slotted page por omissão. Como referido anteriormente a classe `slotted_page` é subtipo do Marrow *array*, e a sua definição está presente na linha 7. Nas linhas 14 e 15 está a definição do grafo que como referido anteriormente é derivada a partir de outra classe: o `vector` da biblioteca standard do C++. Por fim a representação do índice está presente na linha 20.

A figura 3.3 mostra como estão representadas as diferentes estruturas de dados nas memórias da máquina hospedeira e do GPU. Como não há garantias que as páginas sejam alocadas: contiguamente na memória do GPU, é necessário recolher o endereço base das *slotted pages* na memória da GPU, para conseguir aceder à sua informação.

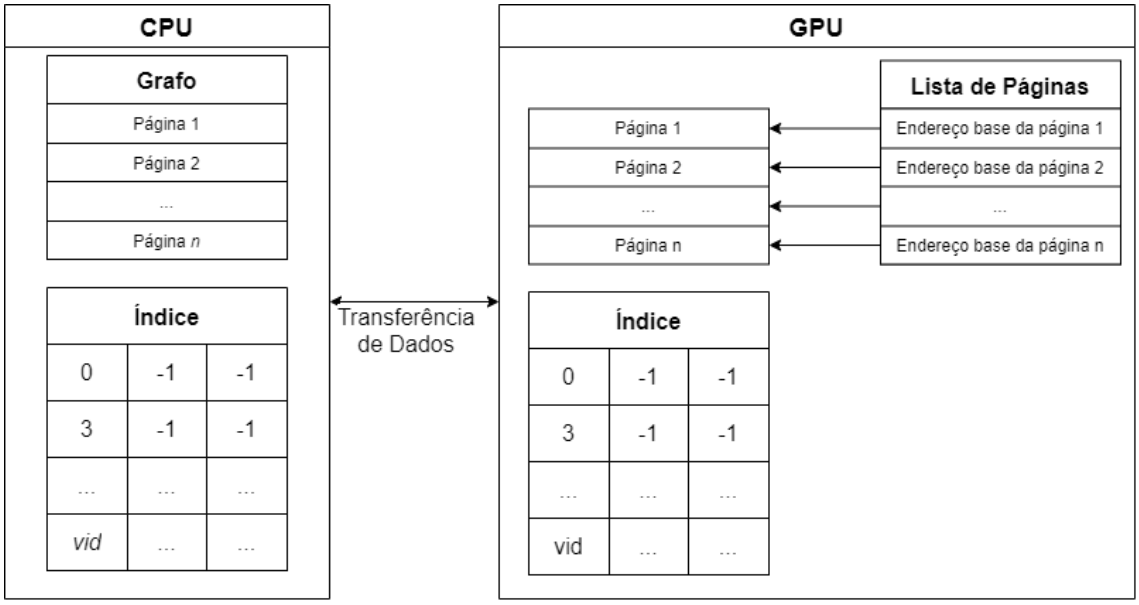

Figura 3.3: Representação das estruturas na memória do sistema

3.3 Manipulação de Grafos

Nesta secção explica-se a implementação das funções que permitem alterar o estado do grafo.

Excepto onde explicitamente indicado o contrário, todas as funções nesta seção são apenas em CPU.

3.3.1 Adição de vértices

A adição de vértices tem de respeitar a ordem crescente na zona das *slots* na página. Esta restrição acaba por beneficiar a procura de uma página para guardar o vértice, pois o novo vértice só pode ser guardado na última página, ou caso esta não tenha espaço suficiente então terá que ser adicionada uma nova página ao grafo e o novo vértice adicionado nesta nova página.

Este procedimento está representado em pseudo-código no algoritmo 1. A escolha do identificador de um vértice é inteiro não negativo cujo valor inicial é "0" e é incrementado em "1" após cada inserção como pode ser observado nas linhas 4 e 5. Na linha 6 é obtida a última página do grafo, dada pela ultima posição do vector *pages*.

De seguida, na linha 7 é verificado se a última página tem espaço livre. O número mínimo de posições livres para que uma página consiga suportar adição de um vértice é três. A primeira posição para o identificador do vértice, a segunda para guardar o índice do início da lista de adjacências e a última para o grau do vértice. No entanto, se considerarmos apenas 3 espaços livres, pode penalizar o desempenho em adições subsequentes, pois ao ocupar completamente uma página não é deixado nenhum espaço para as listas

Algoritmo 1 Adição de um vértice

```
1:  vector<slotted_page> pages                          ▷ páginas do grafo
2:  lastVertexId ∈ N_0                                  ▷ identificador do último vértice adicionado

3:  procedure ADD_VERTEX
4:      vertexId ← lastVertexId
5:      lastVertexId ← lastVertexId + 1
6:      p ← pages.last_page()                           ▷ obter a última página
7:      if not p.has_space() then                       ▷ verificar se a página tem espaço
8:          p ← new slotted_page()
9:          pages.add(p)
10:     end if
11:     p.add_vertex(vertexId)                          ▷ adicionar vértice à página
12: end procedure
```

de adjacências dos vértices presentes crescerem. Para evitar esta situação só é adicionado um vértice a uma *slotted page* que tenha 60% do seu espaço livre. Caso não haja 60% de espaço livre na última página é criada uma nova página (linha 8) e o vértice é adicionado a essa página. No algoritmo 2, da linha 3 à linha 7 apresenta-se o pseudo-código da função $HAS\_SPACE()$ que avalia se a página tem espaço livre. São usados dois valores do cabeçalho, o número de vértices e o tamanho da zona das *records*. O número de vértices multiplicado por 2 resulta no tamanho da zona das *slots*. A soma dos tamanhos das zonas das *records* e *slots* representa o espaço ocupado da página. Para descobrir o espaço ocupado resta subtrair o resultado da soma ao tamanho da página, se este valor for menor que 60% do tamanho da página então a função devolve verdadeiro, e é devolvido falso caso contrário.

Por fim é escrita e atualizada a informação necessária para corretamente adicionar o vértice à página e consequentemente ao grafo. A função *ADD_VERTEX(vid)* no algoritmo 2 descreve este último passo. O vértice é adicionado à última posição da zona das *slots* (linha 11), sendo a última posição calculada por subtrair ao tamanho da página o tamanho da zona das *slots* e por fim subtrair mais um valor. Imediatamente antes do identificador do vértice é guardado o índice do início da lista de adjacências (linha 12). Finalizando, o grau do vértice é colocado a 0 (linha 13) e é atualizada a informação do cabeçalho, incrementando em "1" o número de vértices na página (linha 14) e o tamanho da zona das *records* (linha 15).

3.3.2 Adição de arestas

A adição de arestas é o mecanismo com que o grafo exprime uma relação entre dois vértices. Esta função recebe o identificador de dois vértices, a fonte e o destino, tendo como objetivo adicionar o identificador do vértice de destino à lista de adjacências do vértice fonte. Primeiro é necessário descobrir em que *slotted page* se encontra a lista de adjacências do vértice fonte. Para tal é efetuada uma adaptação da busca binária

Algoritmo 2 Funções de slotted_page

```
1: array<int> page                                          ▷ Conteúdo da página
2: size ∈ N_0                                               ▷ Tamanho da página

3: function HAS_SPACE()
4:     slotsSize ← page[0] × 2          ▷ O número de vértices está guardado no cabeçalho
5:     recordSize ← page[1]                    ▷ Tal como o tamanho da zona das records
6:     return (size - slotsSize + recordSize) < (size × 0.6)
7: end function

8: procedure ADD_VERTEX(vid)
9:     slotsSize ← page[0] × 2
10:    recordSize ← page[1]
11:    page[size - 1 - slotsSize] ← vid                          ▷ Adiciona o vid à page
12:    page[size - 2 - slotsSize] ← recordSize     ▷ Guarda o índice do início da lista de
       adjacências do vid adicionado
13:    page[recordSize] ← 0          ▷ O grau de um novo vértice adicionado é sempre 0
14:    page[0] ← page[0] + 1               ▷ Incrementar o número de vértices na página
15:    page[1] ← page[1] + 1             ▷ Incrementar o tamanho da zona das records
16: end procedure
```

sobre a estrutura auxiliar de indexação, onde em vez de procurarmos o valor exato é procurado um índice cujo identificador do vértice seja maior ou igual à posição atual e simultaneamente menor que a posição seguinte. Após ter sido encontrada a *slotted page* é necessário verificar se o vértice fonte está presente na página, pois a procura sobre o índice não garante que o vértice fonte exista no grafo, pois este pode ter sido removido antes do processo de adição de arestas. No entanto, se ele existir no grafo está definitivamente na *slotted page* encontrada. Se o vértice destino ou o vértice fonte não existirem no grafo a função é abortada e não é feita qualquer alteração ao grafo.

Se a página tiver espaço disponível o vértice é adicionado ao fim da lista de adjacências do vértice fonte. No entanto, pode ser necessário deslocar outras listas dentro da *slotted page*, para criar um espaço à frente da lista desejada, tal como ilustrado na figura 3.4. Caso não exista espaço livre na página é necessário criar uma página imediatamente posterior àquela onde se encontra a lista de adjacências do vértice fonte e podemos encontrar um total de três casos:

1. Encontram-se outras listas imediatamente a seguir à lista de adjacências do vértice fonte. Todas as listas posteriores à do vértice fonte são deslocadas para a nova página juntamente com a informação na zona das *slots* para referenciar essas mesmas listas. A zona das *slots* na nova página precisa de ser atualizada, visto que os índices para o início das listas não são os mesmos na nova página. O identificador vértice destino é adicionado à página original que agora já tem espaço disponível. Esta adição está esquematizada na figura 3.5. É de notar que este procedimento pode também ser aplicado quando o vértice não tem vértices anteriores ao vértice fonte, como

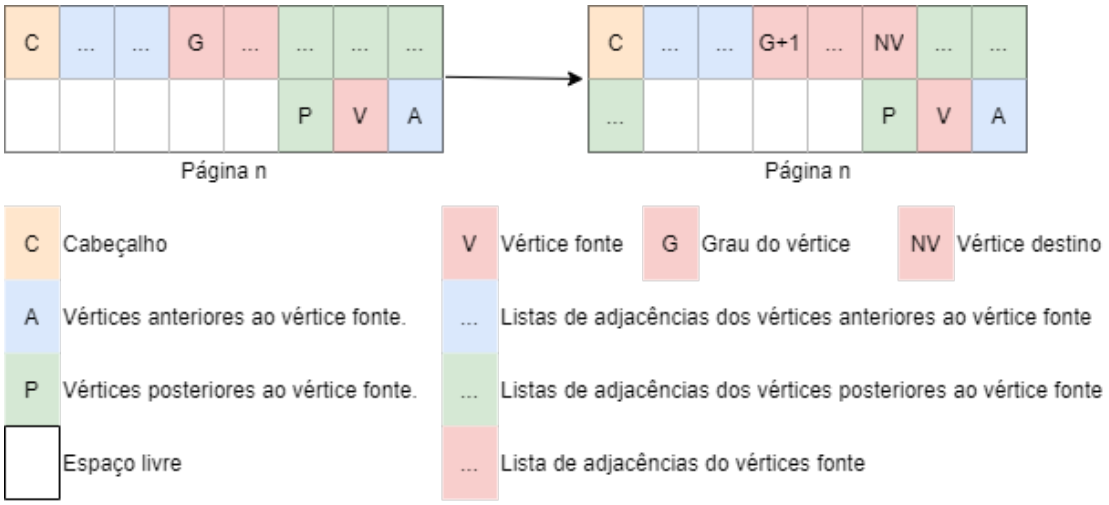

Figura 3.4: Adição com espaço livre na *slotted page*

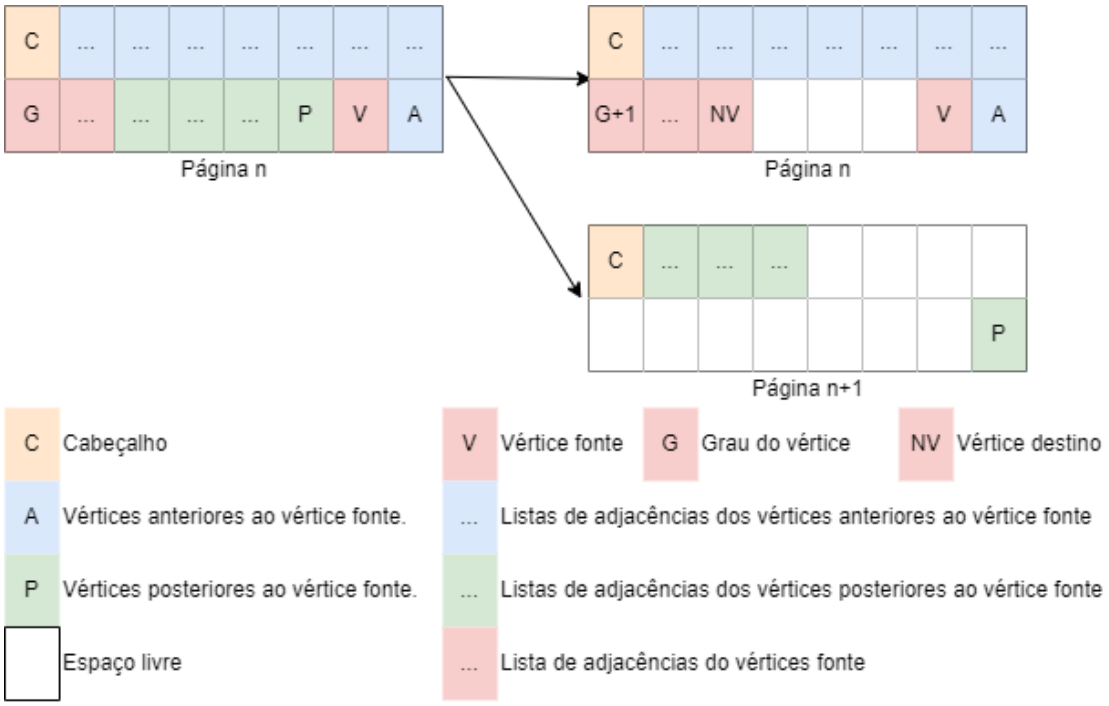

Figura 3.5: Adição sem espaço livre na *slotted page* e com vértices posteriores

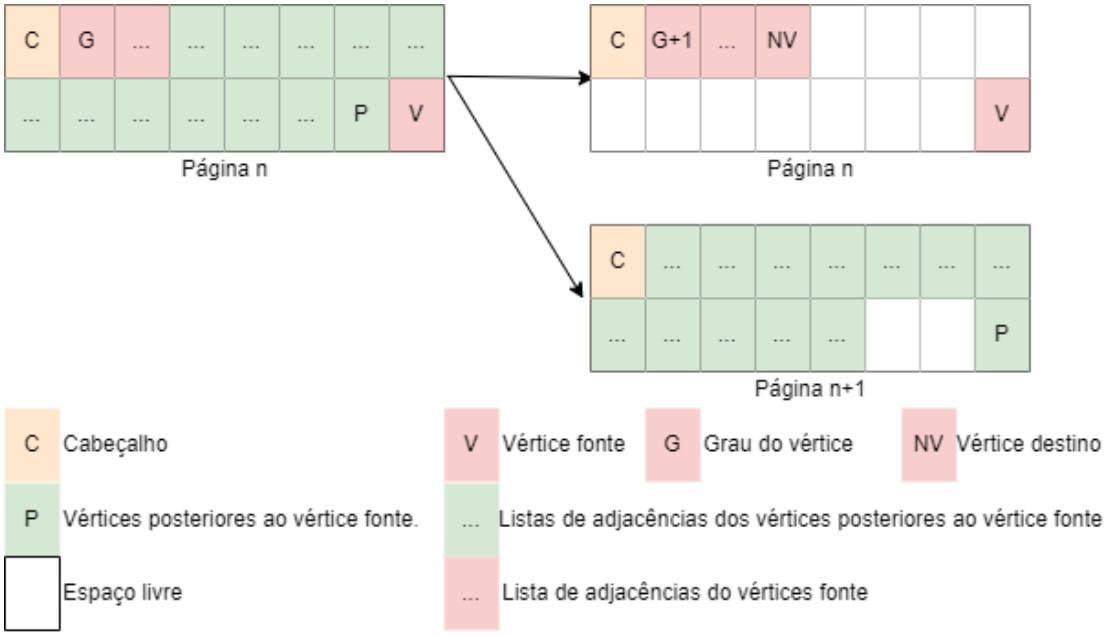

Figura 3.6: Adição sem espaço livre na *slotted page* sem vértices anteriores e com vértices posteriores

ilustrado na figura 3.6.

2. Se a lista de adjacências do vértice fonte for a última da página, então essa lista de adjacências é movida para a nova página e o identificador do vértice destino é adicionado à nova página. Esta adição está esquematizada na figura 3.7.

3. A lista de adjacências do vértice fonte é a única lista na página. Neste caso o identificador do vértice destino é adicionado à nova página. Esta lista de adjacências é agora denominada como uma "grande lista de adjacências". Esta adição está esquematizada na figura 3.8.

Como em todos estes três casos foi criada uma nova página, em todos eles o índice também necessita de crescer para corretamente representar o grafo. A aplicação destes três casos mantém a ordem crescente dos vértices e assegura o mecanismo para a transição entre uma lista de adjacências normal e uma grande lista de adjacências, para que seja possível gradualmente isolar uma lista de adjacências até que esta esteja sozinha numa página. Isto possibilita que, em qualquer estado, o grafo suporte a adição de arestas.

3.3.3 Remoção de arestas

A remoção de uma aresta tem como objetivo retirar um identificador de um vértice da lista de adjacências de outro, semelhante à adição de um vértice. Esta função recebe dois identificadores de vértices, um para a fonte outro para o destino. Tal como adição é necessário encontrar em que página se encontra o vértice fonte, para isto é usado o mesmo

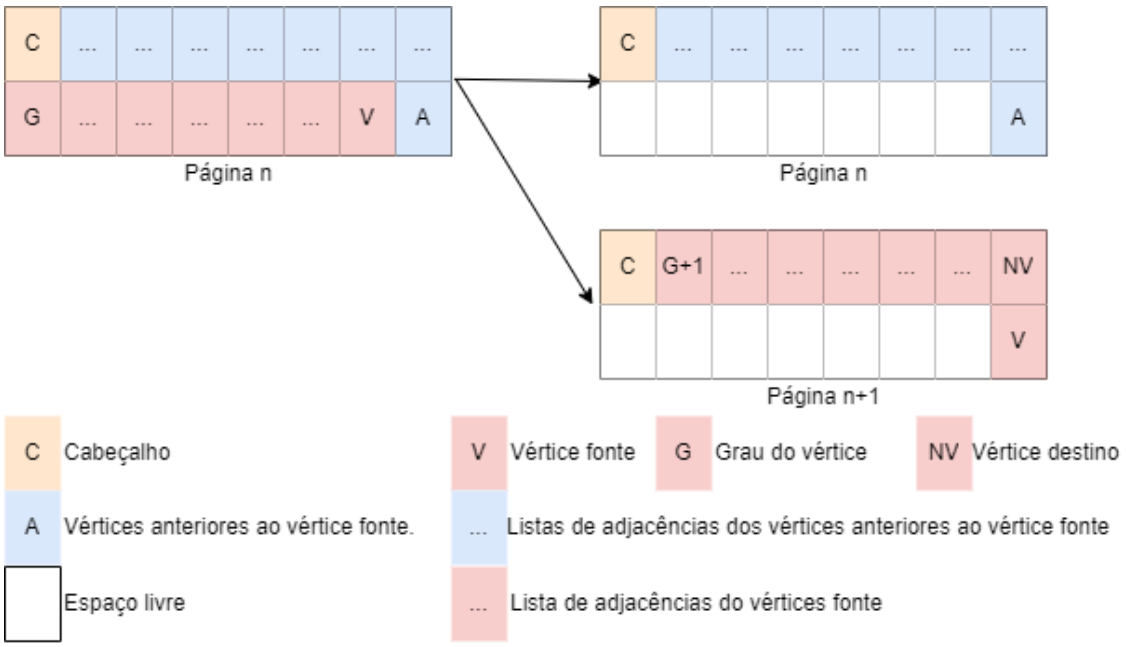

Figura 3.7: Adição sem espaço livre na *slotted page* e sem vértices posteriores

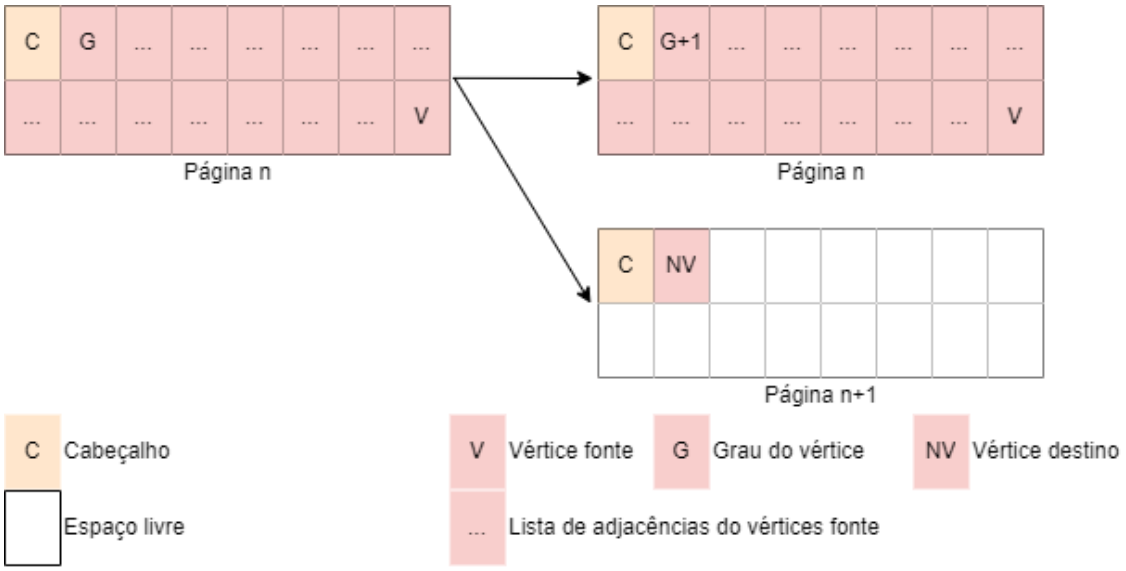

Figura 3.8: Adição sem espaço livre na *slotted page* e sem outros vértices

Listing 3 Segmento de código da função de remoção de arestas

```
1  bool graph::removeEdge(int sourceVid, int destinationVid) {
2      //...
3      int pageIndex = findPage(sourceVid);
4      int startOfAdjacencyList = findIndex(pageIndex, sourceVid);
5      auto& page = DATA[pageIndex];
6      int degree = page[startOfAdjacencyList];
7
8      size_t lowerbound = startOfAdjacencyList + 1;
9      size_t upperbound = lowerbound + degree;
10     marrow::vector<int> adj = page[{lowerbound, upperbound}];
11     marrow::vector<int> bit = adj == destinationVid;
12     coordinate<1> pos = reduce<func::max<int>, reduction_target::indexes>(bit);
13
14     size_t indexOfVidToRemove = pos[0] + lowerbound;
15     //...
16 }
```

método descrito na adição de arestas. Visto que as listas de adjacências não possuem nenhuma ordem, não há maneira de eficientemente encontrar o vértice para ser removido, visto que pode estar em qualquer posição da lista.

Para acelerar esta procura são usadas algumas funcionalidades do Marrow para acelerar a computação com a ajuda do GPU. Parte do código de remoção está presente na listagem 3. Primeiro é preciso descobrir em que página está o vértice (linha 3), o índice do início da lista de adjacências (linha 4) e o grau do vértice (linha 6). A partir destes dados é possível obter os limites da lista de adjacências (linha 8 e 9). De seguida a lista de adjacências do vértice fonte é referenciada com um intervalo (linha 10), deste modo as seguintes operações são direcionadas apenas para a lista de adjacências e não para a página toda. De seguida é feita uma comparação de igualdade em que todas as posições da lista são comparadas com o identificador do vértice de destino, criando um mapa de bits onde a posição do vértice se encontra está a "1" e todas as outras a "0" (linha 11). Para saber em que índice se encontra o 1 é feita uma redução onde se quer descobrir a posição do valor máximo do mapa de bits (linha 12). Se a lista for uma grande lista de adjacências, são usados os mesmos passos iterativamente até ser encontrada a página onde se encontra o identificador do vértice de destino.

Após conhecermos a posição do vértice a remover, este é trocado com o vértice que está na última posição da lista e o grau do vértice da fonte é diminuído numa unidade. Caso existam listas de adjacências posteriores à editada, são movidas de modo a deixar a zona das *records* sem espaços inutilizados entre as diferentes listas. No caso de ser uma grande lista de adjacências é possível que a última página da lista fique apenas com o vértice eliminado, neste caso, essa página é apagada.

Listing 4 Edição de uma aresta

```
1 bool graph::editEdge(int sourceVid, int oldDest, int newDest) {
2     bool removeEdge = this->removeEdge(sourceVid, oldDest);
3     bool addEdge = this->addEdge(sourceVid, newDest);
4     return removeEdge && addEdge;
5 }
```

3.3.4 Remoção de vértices

A remoção do vértice do grafo elimina também a sua lista de adjacências e não é verificado se o vértice a remover está presente noutras listas de adjacências. Esta funcionalidade pode ser obtida visitando todas as listas de adjacências e removendo o vértice quando presente, visto que isto seria uma operação computacionalmente extensa não está incorporada diretamente na função de remoção.

Para removermos a lista de adjacências do vértice é chamada iterativamente a função de remoção de arestas para todos os valores da lista de adjacências, por fim o identificador do vértice e o índice para o início da lista de adjacências guardado na zona das *slots* é sobreposto com vértices que estejam diretamente a seguir. Se o vértice a ser removido for o primeiro da página a estrutura auxiliar tem que ser atualizada e se a página ficar sem vértices é apagada do grafo.

3.3.5 Edição de arestas

A função recebe três argumentos, o identificador do vértice fonte, o identificador do vértice de destino a ser substituído e o novo vértice destino. A edição de vértices é obtida ao utilizar a função de remoção de uma aresta (secção 3.3.3) seguido da adição de uma nova aresta (secção 3.3.2).

3.3.6 Sincronização com a GPU

Como referido anteriormente, o Marrow orquestra a transmissão de dados quando necessários para o GPU. Isto permite que possa haver várias operações de adição ou remoção de vértices e adição ou edição de arestas, sem que seja necessário transmitir os dados para o GPU após uma alteração. A única exceção é a remoção de arestas que por usar funções do Marrow que são exectuadas em GPU vai causar com que algumas páginas sejam transmitidas para o GPU, durante a execução da função.

3.4 Processamento de Grafos

Na secção 2.3.2, foram enumeradas operações primitivas de modo a tornar possível a execução de algoritmos em grafos. Neste capítulo analisa-se com maior detalhe essas primitivas para compreender o que Marrow já possui e o que foi adicionado. Estas operações

são executadas em GPU.

3.4.1 Compute

A operação *compute* aplica uma função definida pelo utilizador a todos os elementos da fronteira, o tamanho do resultado (*output*) é igual ao tamanho do argumento de entrada (input). Uma fronteira é definida como uma coleção de identificadores de vértices. O *compute* não é mais do que uma operação de *map* que está disponível no Marrow.

3.4.2 Advance

O *advance* é uma operação que obtém para uma dada lista de identificadores de vértices as suas respetivas listas de adjacências. Como descrito no algoritmo 3, tanto o input (linha 1) como o output (linha 2) são uma coleção de vértices. No output as listas de adjacências dos vértices dados como input, estão na mesma ordem do que o input, primeiro a lista do *vid1* depois do *vid2*, etc. O output pode conter vértices repetidos, devido ao mesmo vértice fazer simultaneamente parte da lista de adjacências de mais do que um vértice dado no input.

Algoritmo 3 Advance

```
1:  Graph graph
2:  list<int> frontier ← {vid1, vid2, ..., vidn}
3:  list<int> result ← Advance(frontier, graph)

4:  function ADVANCE(fronteir, graph)
5:      pages ← graph_find_page(fronteir, graph.get_index())      ▷ Índices das páginas
    onde estão os vértices da fronteira
6:      index ← graph_find_index(fronteir, pages, graph)   ▷ Índices do inicio das listas
    de adjacências
7:      degrees ←graph_get_degree(index, pages)          ▷ Graus dos vértices da fronteira
8:      scan_degrees ← scan(degrees)   ▷ Scan (exclusivo) aplicado aos graus dos vértices
9:      sum_degrees ← degrees.last() + scan_degrees.last()                ▷ Soma dos graus
10:     if sum_degrees == 0 then                    ▷ Se for 0, não há vértices a obter
11:         return new list<int>()                                        ▷ Lista vazia
12:     end if
13:     aux_index ← graph_binary_search_indexes(scan_degrees, size)
14:     aux_values ← graph_binary_search_values(aux_index, scan_degrees)
15:     result ← graph_expand_vertex(aux_index, index, aux_values, pages, graph)
16:     return result
17: end function
```

A implementação do operador *advance* não é trivial devido à configuração dinâmica do grafo. Os vértices podem estar em qualquer página, o início da lista de adjacências de um vértice pode estar em qualquer posição na página e a lista pode inclusive estender-se por várias páginas.

Listing 5 Código em OpenCL da função aplicada no *map* `graph_find_page`

```
1  int graph_find_page(int vertexId, global int* spIndexer,
2  unsigned long numberOfPages){
3    size_t high = numberOfPages - 1;
4    size_t low = 0;
5
6    while(low <= high){
7      size_t middle = low + ((high - low)/2);
8
9      //Valores guardados para reduzir os acessos à memória global
10     size_t middleIndex = middle * 3;
11     int m3 = spIndexer[middleIndex];
12     int m3p1 = spIndexer[middleIndex + 1];
13
14     if(( (middle == numberOfPages - 1 && vertexId >= m3) ||
15     (m3p1 == -1 && vertexId >= m3 && vertexId < spIndexer[(middle + 1) * 3]) ||
16     (m3p1 != -1 && vertexId == m3)))
17       return m3p1 == -1 ? middle : middle - m3p1;
18     if (m3 < vertexId)
19       low = middle + 1;
20     else
21       high = middle - 1;
22   }
23   return -1;
24 }
```

Outro ponto importante é como é feita a distribuição de carga durante o preenchimento do resultado final. Uma vez que o objetivo é cada vértice recolher a sua lista de adjacências faz sentido numa primeira análise distribuir um vértice por *thread*, sendo esta responsável por recolher na totalidade a lista de adjacências de cada vértice. No entanto esta abordagem revela-se ineficiente se considerarmos que é comum haver uma grande discrepância entre os graus de cada vértice no grafo. Por exemplo, a uma *thread* poderia ser atribuído um vértice com um grau de 1000 e a outra *thread* um vértice com um grau de 2. Para evitar esta má distribuição foram implementados passos extra para balancear a carga entre os *threads*, de forma a que a cada *thread* seja atribuída a tarefa de recolher apenas um vértice da lista de adjacências.

O algoritmo 3 mostra os passos necessários para implementar o operador *advance*. Na totalidade são necessários 8 passos, sendo que os passos da linha 13 e 14 foram adicionados para melhorar o balanço de carga.

De seguida vamos explicar o algoritmo passo a passo.

graph_find_page A função `graph_find_page(map)` aplica um mapa Marrow a uma coleção de identificadores de vértices para encontrar os índices das páginas onde os

mesmos se encontram.

No GPU a função aplicada pelo *map* recebe um único vértice e é responsável por encontrar a página onde este se encontra. A listagem 5 apresenta a implementação realizada em OpenCL.

O algoritmo tira partido da ordem crescente do grafo (e consequentemente do índice auxiliar), para acelerar a procura aplicando uma adaptação da busca binária ao índice, relembrando que o índice guarda o primeiro identificador do vértice de cada página. Esta adaptação tem em consideração a estrutura do índice e não procura um valor específico, mas sim uma posição em que o identificador do vértice seja igual ou maior do que a posição atual e simultaneamente menor do que a posição anterior (linha 15). No entanto, se tivermos a considerar a última posição, só é necessário verificar se o vértice que estamos a procurar é maior ou igual do que a última posição, visto que não há mais páginas a verificar (linha 14). Também é necessário endereçar o caso especial de quando as páginas fazem parte de uma grande lista de adjacências, pois nesse caso só temos que fazer uma comparação de igualdade (linha 16). É, portanto, retornada a posição da primeira página da grande lista de adjacências.

graph_find_index A função `graph_find_index(map)` aplica um *map* Marrow a uma coleção de identificadores de vértices e as páginas onde eles estão localizados. A função aplicada é responsável por retornar o índice da página que indica o início da lista de adjacências do seu respetivo vértice. A listagem 6 apresenta uma implementação OpenCL.

Neste algoritmo também é possível tirar partido da ordem crescente do grafo e aplicar uma busca binária à zona das slots procurando a posição do identificador do vértice, pois o índice do início da lista de adjacências está diretamente antes. Mas ainda é possível melhorar esta busca, pois se a diferença do último identificador de vértice na página com o primeiro for igual ao número de vértices na página, podemos concluir que nenhum vértice foi removido na página e automaticamente sabemos a posição de todos os vértices na zona das *slots*. Esta propriedade é aplicável a qualquer subseção continua na zona das *slots*, o que quer dizer que podemos testar esta condição em todos os ciclos da busca binária, acelerando ainda mais o algoritmo.

graph_get_degree A função `graph_find_index(map)`, recebe os índices do início das listas de adjacências e as suas respetivas páginas. Cada *thread* é responsável por retornar um grau da respetiva lista de adjacências (e por extensão do vértice que este representa). Esta função é simples pois o grau é o primeiro valor da lista de adjacências. A função está presente na listagem 7.

scan dos graus. É feito o *scan* dos graus (o scan é explicado em maior detalhe na secção 4.1). O resultado do *scan* é um resultado intermédio que vai ser usado nos passos seguintes. O *scan* também diferencia os limites das diferentes listas de adjacências no resultado final.

Listing 6 Código em OpenCL do map graph_get_index

```
#define SLOTTED_PAGE_SIZE 1024

int graph_get_index(const int vertexId, global int* page) {
  int high = page[0] - 1;
  int low = 0;

  while(low <= high){
    int highPos = SLOTTED_PAGE_SIZE - 1 - (high * 2);
    int lowPos = SLOTTED_PAGE_SIZE - 1 - (low * 2);
    int pl = page[lowPos];

    if(page[highPos] - pl == high - low)
      return page[(lowPos - (vertexId - pl)*2) - 1];

    int middle =  low + (high - low) / 2;
    int middlePos = SLOTTED_PAGE_SIZE - 1 - (middle * 2);
    int pm = page[middlePos];

    if(pm == vertexId)
      return page[middlePos-1];
    if (pm > vertexId)
      high = middle - 1;
    else
      low = middle + 1;
  }
  return -1;
}
```

Listing 7 Código em OpenCL do map graph_get_degree

```
int graph_get_degree(int pos, global int* page) {
  return page[pos];
}
```

O último valor do *scan* somado com o último valor da lista de graus dá o número de elementos no resultado final. Se a soma for zero, então o *advance* pode devolver uma lista vazia e terminar neste passo.

graph_binary_search_indexes A função graph_binary_search_indexes aplica um *map* Marrow que efetua uma adaptação da busca binária entre todos os valores inteiros entre 0 e o resultado da soma dos graus sobre o resultado do scan dos graus. A função aplicada pelo mapa recebe um inteiro e devolve o índice que cumpra a condição do seu que seja maior ou igual do que o índice atual e simultaneamente menor que a posição seguinte. A

Listing 8 Código em OpenCL do map graph_binary_search_indexes

```
1  int graph_binary_search_indexes(int toFind, global int* scan,
2  const unsigned long size) {
3    size_t high = size - 1;
4    size_t low = 0;
5
6    while(low <= high){
7      size_t middle = low + ((high - low)/2);
8      int sm = scan[middle];
9
10     if ((middle == size - 1) || ((toFind >= sm) && (toFind < scan[middle+1])))
11       return middle;
12     if (sm <= toFind)
13       low = middle + 1;
14     else
15       high = middle - 1;
16   }
17
18   return -1;
19 }
```

Listing 9 Código em OpenCL do map graph_binary_search_values

```
1  int graph_binary_search_values(int index, global int* buffer) {
2    return buffer[index];
3  }
```

função está implementada na listagem 8 e a condição descrita está presente na linha 10.

graph_binary_search_values A função graph_binary_search_value também aplica um *map* Marrow. Usa o resultado do passo anterior como um índice para retornar o valor correspondente do *scan* dos graus. A função aplicada é responsável por devolver o valor correspondente a um índice. Esta função está presente na listagem 9.

graph_expand_vertex A função graph_expand_vertex (*map*), usa o resultado dos 2 passos anteriores para devolver os vértices das listas de adjacências. Cada *thread* tem apenas a responsabilidade de devolver um valor. A função está presente na listagem 10 e esta tem em consideração que a lista de adjacências pode estar distribuída por diferentes páginas e ajusta automaticamente os indicies para devolver o valor correto. Em primeiro lugar a função calcula um deslocamento, este valor somado ao início da lista de adjacências vai dar o índice correto na página para ser devolvido o identificador do vértice da lista de adjacências. Se este deslocamento for maior que o tamanho da página (menos o cabeçalho e 2 posições para a zona das *slots*) significa que esta lista se trata de uma grande lista e

Listing 10 Código em OpenCL do map graph_expand_vertex

```
1  #define SLOTTED_PAGE_SIZE 1024
2
3  int graph_expand_vertex(int index, global int* vidLocation, global int* auxValues,
4  global int* pages, global unsigned long* address) {
5    int offset = get_global_id(0) - auxValues[get_global_id(0)];
6
7    int pagePos = offset + 1 > SLOTTED_PAGE_SIZE - 6 ?
8    (offset - (SLOTTED_PAGE_SIZE - 6))/(SLOTTED_PAGE_SIZE - 3) + 1 + pages[index] :
9    pages[index];
10
11   global int* page = ( global int* ) address[pagePos];
12
13   int newOffset = offset + 1 > SLOTTED_PAGE_SIZE - 6 ?
14   ((offset - (SLOTTED_PAGE_SIZE - 6)) % (SLOTTED_PAGE_SIZE - 3)) + 3 :
15   offset + vidLocation[index] + 1;
16
17   return page[newOffset];
18 }
```

que o vértice está noutra página. No entanto a divisão inteira do deslocamento menos o tamanho da página (menos o cabeçalho e a zona das *slots*) com o tamanho da página menos o cabeçalho, devolve o deslocamento do número de páginas para o vértice correto. A mesma lógica pode ser estendida para calcular o deslocamento correto caso o vértice faça parte de uma grande lista de adjacências, só que em vez da divisão é aplicado o operador para o resto da divisão inteira.

Resolução do problema de distribuição de carga. Em conjunto, os passos do *scan* dos graus, do `graph_binary_search_indexes` e do `graph_binary_search_values` criam uma lista de valores que correspondem ao deslocamento a ser adicionado ao início da lista de adjacências, de modo a ser referenciado apenas um valor da lista. Desta forma é possível atribuir a uma *thread* o trabalho de devolver apenas um valor, eliminando o problema de distribuição de carga. A figura 3.9 ilustra como é que este valor é calculado. O ultimo cálculo na ilustração está presente na linha 5 da listagem 10.

3.4.3 Filter

A operação *filter* filtra a fronteira com base numa condição definida pelo programador. Cada valor na fronteira de input pode originar um ou zero valores na fronteira de output.

Dado que a operação de filtro é uma operação genérica, cuja utilização pode ser útil em muitos contextos para além do processamento de grafos, optou-se por incluir a operação diretamente no Marrow, assim permitindo o seu uso generalizado.

A implementação é detalhada na secção 4.2.

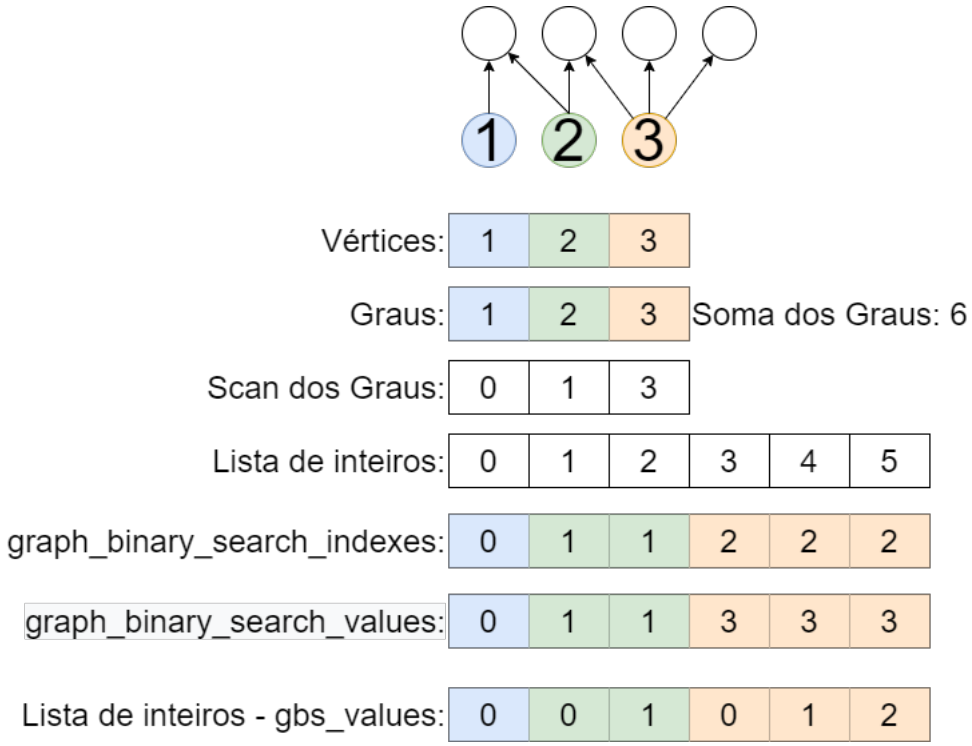

Figura 3.9: Esquematização das operações necessárias para resolver o problema de balanceamento de carga

3.4.4 Segmented Intersection

A operação *segmented intersection* recebe duas fronteiras com o mesmo tamanho e, para cada par de vértices, computa as intersecções (os vértices em comum) das suas expansões. Para tal recebe as duas fronteiras (listas com identificadores de vértices) com o mesmo tamanho, aplica a operação *advance* a ambas as listas e calcula os vértices em comum para cada segmento, sendo os segmentos o resultado da expansão para cada vértice na mesma posição das listas iniciais, como exemplificado na figura 3.10. Nesta figura cada segmento é representado por uma cor diferente, é de notar que apenas as listas iniciais precisam de ter o mesmo tamanho (para que haja o mesmo número de segmentos), após a aplicação do *advance* as listas não precisam de ter o mesmo tamanho.

A função utiliza alguns resultados calculados durante a aplicação do *advance*. Utiliza o *scan* dos graus da segunda lista de modo a calcular os índices de cada segmento e utiliza o resultado da função `graph_binary_search_indexes` sobre a primeira lista de modo a referenciar os segmentos. A função final foi implementada como um *scan*, onde cada vértice da primeira lista faz a pesquisa nos respetivos segmentos da segunda. A função está presente na listagem 11.

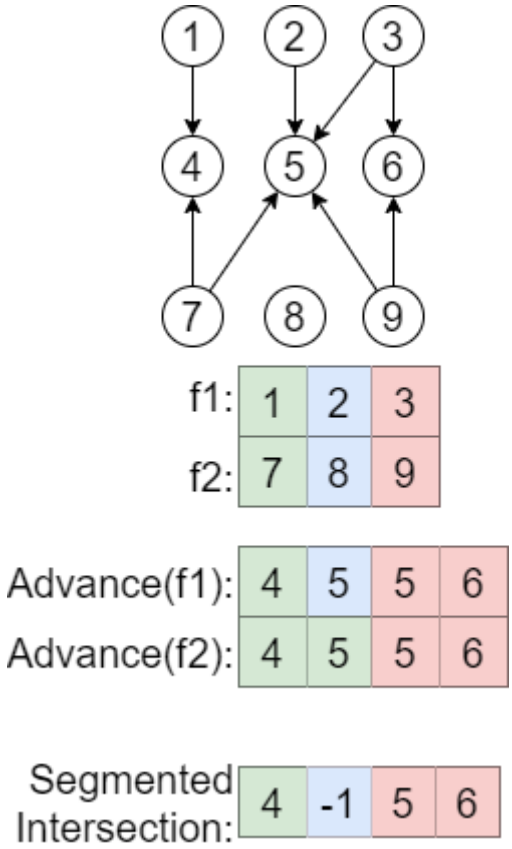

Figura 3.10: Exemplo da aplicação do Segmented Intersection. *f1* e *f2* representam fronteiras.

Listing 11 Código em OpenCL do map segmented_intersection

```
1  int segmented_intersection(int toFind, global int* lookup, global int* scanLookup,
2  int binaryIndex, unsigned long lookupSize, unsigned long scanSize){
3      int start = scanLookup[binaryIndex];
4      int end = binaryIndex + 1 >= scanSize ? lookupSize : scanLookup[binaryIndex+1];
5
6      for(int i=start; i < end; i++)
7          if(toFind == lookup[i])
8              return toFind;
9
10     return -1;
11 }
```

3.5 Conclusão

Devido à organização específica e necessária de uma *slotted page* é difícil utilizar as funcionalidades do Marrow para acelerar a sua construção. A utilização do range permite fazer procuras mais eficientes às listas de adjacências, mas é apenas possível a sua utilização em edições ou remoções. As funções de adição, que são as mais usadas na geração de qualquer grafo, são operações computacionalmente caras, pois precisam de mover outras listas de adjacências dentro da própria página para acomodar adições a uma lista de adjacências ou movidas para uma nova página para manter a ordem crescente do grafo. No entanto esta solução oferece o pretendido - uma forma de criar um grafo sem saber nenhum detalhe antes, seja o número total de vértices ou número total de arestas e suportar adições, remoções ou edições em qualquer estado.

No que faz parte da implementação das funções para o processamento de grafos, o Marrow possui imediatamente funcionalidades que correspondem ao desejado para implementar a maioria das operações, com a exceção do *Scan*. A única limitação foi na chamada das funções geradas pelo Marrow em OpenCL no operador *Advance*, que não possui a possibilidade de endereçar as páginas fazendo uso de outra estrutura de dados. Para contornar esta limitação alguns *kernels* foram editados posteriormente a serem gerados pelo Marrow, é de notar que estas alterações foram feitas de modo a diminuir os tempos de execução, ao estruturar os dados de uma maneira diferente seria possível ter o mesmo comportamento sem necessitar de alterar os *kernels* manualmente, mas teria que ser necessário a adição de mais três passos: dois para construir de maneira diferente a lista de páginas, que iria precisar de transmitir dados da GPU para o CPU e a execução de um *map* extra para calcular o índice das páginas antes da expansão. Estas alterações causariam uma sobrecarga significativa numa função fundamental do grafo e por isso esses três passos extra não foram implementados. Isto mostra simultaneamente uma versatilidade no Marrow, que permite editar os *kernels* contornado este problema e a falta de uma customização mais granular nas chamadas das funções automaticamente geradas.

4

Adição de Novas Funcionalidades ao Marrow

Neste capítulo é descrito como foram implementadas novas funcionalidades ao Marrow. Estas funcionalidades foram necessárias para a implementação do grafo dinâmico, mas o seu uso não está condicionado ao paradigma de processamento de grafos. O capítulo começa com a descrição da implementação do *scan* (secção 4.1), seguindo-se a descrição da implementação do filtro (secção 4.2) e, por último, descreve-se o suporte para a computação sobre *Containers* de Apontadores (secção 4.3).

4.1 Scan

O *scan* é um esqueleto algorítmico simples e com uma utilização alargada em diversos contextos de aplicação. O *scan* consiste em aplicar uma operação binária, desde que esta tenha as propriedades de comutação e associação (como por exemplo uma soma ou multiplicação), cumulativamente. Isto faz com que em cada posição do resultado tenho sido aplicada a operação conjuntamente a todos os elementos anteriores. Tipicamente há o *scan* exclusivo, que começa a 0 e exclui a última aplicação do operador e o inclusivo que mantém essa última aplicação. O exemplo de um *scan* inclusivo e exclusivo em que o seu operador é uma soma pode ser visto na figura 4.1.

No Marrow foi implementado o *scan* exclusivo em GPU, tem como base algoritmo usado é o descrito por Mark Harris, Shubhabrata Sengupta e John D. Owens em "GPU Gems 3" [14] que realiza um *scan* para uma soma sobre inteiros. O Marrow possui funcionalidades que permitem definir o operador e o tipo do *input*.

Para alcançar o *scan* é necessário dois passos. O primeiro passo divide o *input* em vários segmentos e é aplicado o *scan* a cada segmento. O segundo processa esses segmentos obtendo um *scan* global.

Numa primeira fase são transferidos segmentos do vetor original para memória local do grupo, e é aplicado o *scan* à memória local, visto que o *scan* envolve uma série de acessos à memória, seria pouco eficiente fazer estas operações em memória global. A listagem 12 mostra o código em OpenCL desta transferência (linhas 19 e 20), é de notar

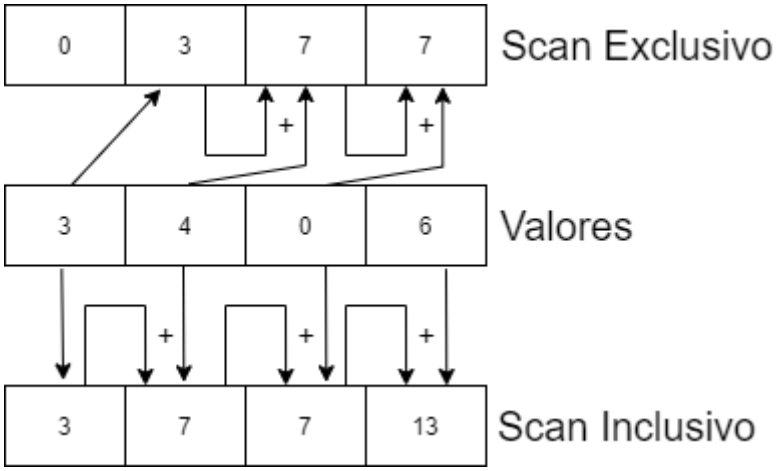

Figura 4.1: Diferenças entre o *scan* exclusivo e o *scan* inclusivo

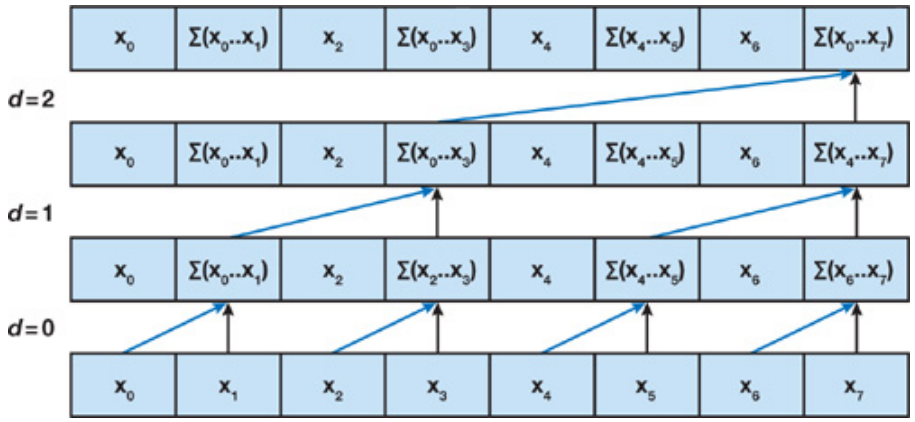

Figura 4.2: Esquematização das operações feitas na fase *up-sweep* [14]

que cada *thread* copia dois valores e para índices maiores que o tamanho do vetor original é colocado um zero. Isto é feito porque as fases seguintes precisam que o tamanho da memória local seja uma potência de dois, no entanto para poder ser aplicado o *scan* a estruturas de qualquer tamanho é colocado zeros por estes não afetarem as seguintes operações de adição. Também é importante salientar que na linha 1 e 2 são definidas duas macros geradas automaticamente pelo Marrow. A macro na linha 1 define o tipo dos dados, neste caso inteiros e a linha dois define o operador, neste caso o operador de adição.

De seguida é efetuada a fase *up-sweep* à memória local (figura 4.2). Nesta fase a memória local é endereçada como se tratasse de uma árvore binária, com o objetivo de ajudar a determinar o que cada *thread* tem que fazer em cada passo. A árvore é percorrida das folhas até à raiz, sendo a adição dos valores dos nós filhos guardados no nó pai. No final desta fase o resultado de todas as somas fica guardado na raiz. A listagem 13 mostra o código em OpenCL desta fase.

Após a fase *up-sweep* ser concluída na última posição (a raiz da árvore) da memória

Listing 12 Carregamento dos dados para memória local

```
#define T int
#define Operation(A, B) (A + B)

kernel void marrow_kernel_one(global T *g_odata, global T *aux, global T
↪  *g_idata, local T *tmp, const unsigned long  n) {
  //...

  size_t globalId = get_global_id(0);
  size_t localId = get_local_id(0);
  size_t localSize = get_local_size(0);
  size_t groupId = get_group_id(0);
  size_t groupSize = get_num_groups(0);

  size_t local2t = localSize * 2;
  size_t gid2 = globalId * 2;
  size_t gid2p1 = gid2 + 1;
  size_t lid2 = localId * 2;
  size_t lid2p1 = lid2 + 1;

  tmp[lid2] = gid2 < n ? g_idata[gid2] : 0;
  tmp[lid2p1] = gid2p1 < n ? g_idata[gid2p1] : 0;
  //...
```

local encontra-se o resultado da soma de todos os elementos desse segmento, esse valor é guardado numa estrutura auxiliar para ser utilizado posteriormente e depois é colocado um zero na raiz da árvore, iniciando-se a fase *down-sweep* (figura 4.3). De seguida a árvore é percorrida da raiz até às folhas. A cada passo o valor de cada nó é passado para o filho esquerdo, o valor do filho direito é a soma do valor antigo do filho esquerdo com o valor do nó pai. A listagem 14 mostra em OpenCL os passos desta fase.

Ao concluir-se a fase *up-sweep* e *down-sweep* foi aplicado o *scan* apenas em segmentos do vetor original e não ao vetor original como um todo. Para se obter um *scan* 'global' é aplicado o *scan* à estrutura auxiliar, esta estrutura é pequena o suficiente para caber totalmente em memória local, por isso quando concluído o *scan*, este foi aplicado totalmente à estrutura auxiliar. Por fim executa-se a última fase, onde cada posição da estrutura auxiliar é somada a todos os elementos de um segmento. A listagem 15 mostra em OpenCL o código deste último passo.

4.2 Filter

O filtro pode ser dividido em duas fases distintas, uma fase onde a condição é avaliada e outra fase onde apenas valores desejados são transferidos para o resultado final. O

Listing 13 Código em OpenCL da fase *up-sweep*

```
1  //...
2  int offset = 1;
3  for (int d = localSize; d > 0; d = d>>1){
4      barrier(CLK_LOCAL_MEM_FENCE);
5      if (localId < d){
6          int ai = offset*(2*localId+1)-1;
7          int bi = offset*(2*localId+2)-1;
8          tmp[bi] = Operation(tmp[ai], tmp[bi]) ;
9      }
10     offset = offset << 1;
11 }
12 //...
```

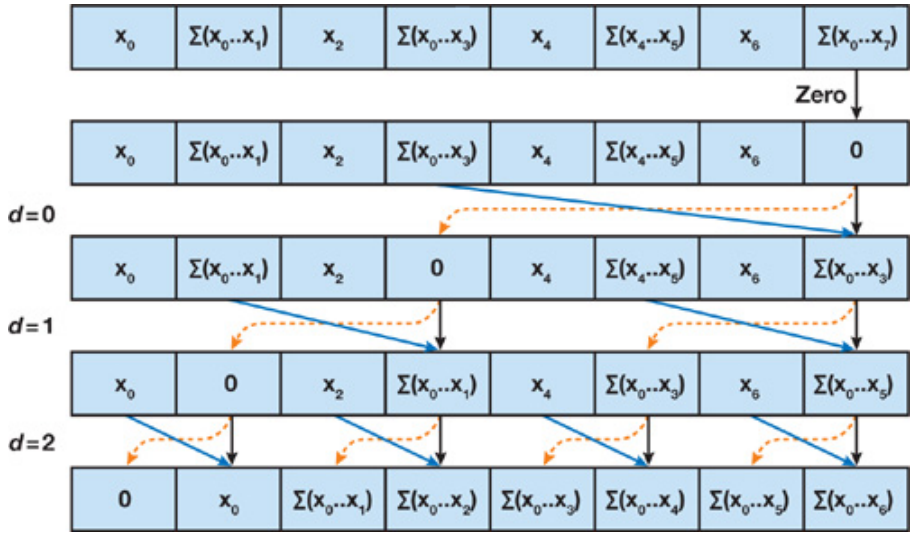

Figura 4.3: Esquematização das operações feitas na fase *down-sweep* [14]

Marrow já possui um mecanismo de avaliar expressões a partir da função *conditional*, esta função permite não só avaliar uma expressão como customizar o valor a ser devolvido caso a expressão seja avaliada como verdadeira ou falsa. Deste modo é possível criar um mapa de bits onde "1" significa que a expressão foi avaliada como verdadeira e "0" caso contrário. Este mapa de bits é usado para descobrir o tamanho da estrutura final filtrada, este valor é calculado a somar todos os valores do mapa de bits. O mapa de bits também é usado para calcular os índices onde são guardados os valores filtrados, para tal é necessário aplicar o *scan* ao mapa de bits. A ilustração 4.5 esquematiza as operações necessárias para o filtro funcionar e a listagem 16 mostra o *kernel* implementado em OpenCL.

Listing 14 Código em OpenCL da fase *down-sweep*

```
//...
barrier(CLK_LOCAL_MEM_FENCE);

if(localId == 0){
    aux[groupId] = tmp[local2t - 1];
    tmp[local2t - 1] = 0;
}

for (int d = 1; d < local2t; d = d<<1){
    offset =  offset >> 1;
    barrier(CLK_LOCAL_MEM_FENCE);
    if (localId < d){
        int ai = offset*(2*localId+1)-1;
        int bi = offset*(2*localId+2)-1;
        T t = tmp[ai];
        tmp[ai] = tmp[bi];
        tmp[bi] = Operation(t, tmp[bi]);
    }
}
//...
```

Listing 15 Código em OpenCL da última fase

```
kernel void marrow_kernel_two(global T *g_out, global T *aux,
const unsigned long n) {
        size_t globalId = get_global_id(0);
        size_t groupId = get_group_id(0);
        size_t localSize = get_local_size(0);

        if(globalId >= n) return;
        g_out[globalId+localSize] += aux[groupId+1];
}
```

4.3 Containers de Apontadores

Até ao início deste trabalho, o Marrow apenas suportava *containers* de valores primitivos ou de outros *containers*.

Para melhor suportar o processamento de grafos, implementámos *containers* de apontadores. Tal permite eficientemente aglomerar todas as páginas numa única estrutura para que nesta estrutura sejam aplicadas funções ao grafo como um todo. Sem esta implementação a alternativa seria aplicar a função página a página o que causaria uma sobrecarga significativa no desempenho, não sendo compatível com os objetivos desta tese.

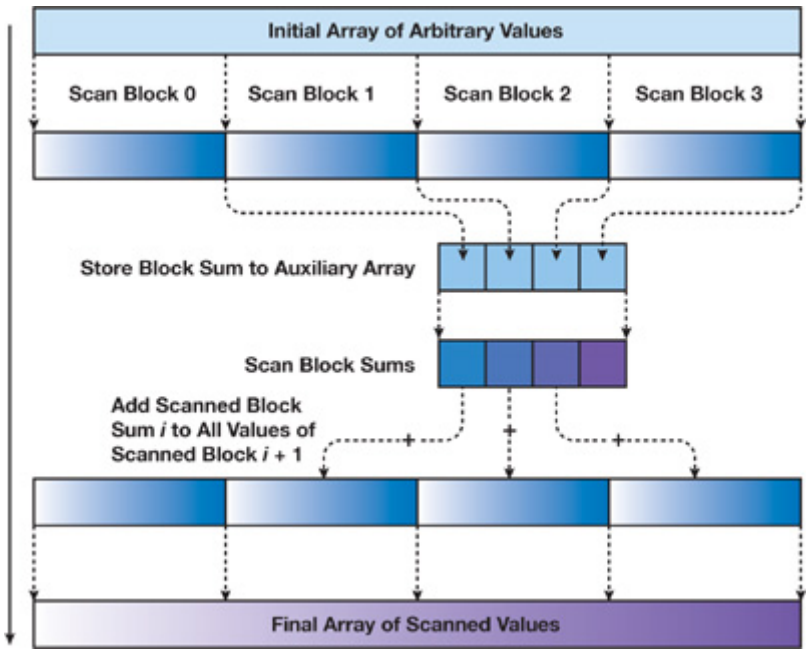

Figura 4.4: Esquematização das operações feitas para obter uma soma global [14]

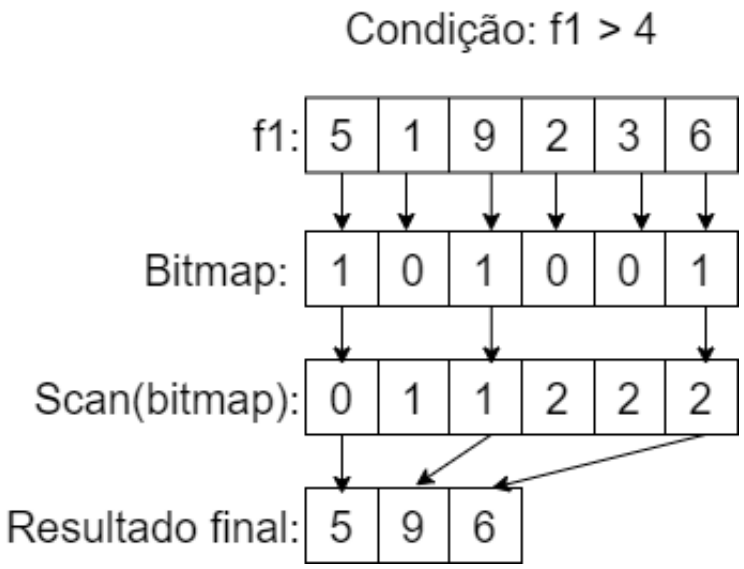

Figura 4.5: Esquematização das operações feitas no filtro

Os *containers* de apontadores funcionam ao guardar o endereço base das estruturas de dados guardados. Por isto, este tipo de *container* funciona de uma maneira diferente do que as outras estruturas de dados, em vez de ter os mesmos dados replicados tanto em CPU como em GPU a lista necessita de ter informação diferente nos dois lados, pois o endereço base de uma estrutura de dados não é o mesmo no CPU e em GPU.

Para descobrir o endereço base de uma estrutura de dados em CPU é feito diretamente com o uso do operador &, para a GPU é necessária a execução de um *kernel* para cada

Listing 16 Código em OpenCL do filtro

```
kernel void marrow_kernel(global T *g_ibit, global T * g_odata,
global T *g_iscan, global T *g_idata, const unsigned long n) {

    size_t globalId = get_global_id(0);
    if(globalId < n && g_ibit[globalId])
        g_odata[g_iscan[globalId]] = g_idata[globalId];
}
```

Listing 17 Código em OpenCL usado para obter o endereço base de uma estrutura de dados

```
#define ADDR unsigned long

kernel void marrow_kernel(global ADDR* out, global int* in,
unsigned long index) {
  const size_t tid  = get_global_id(0);
  if (tid == 0)
    out[index] = (ADDR) in;
}
```

elemento da lista, isto tem um custo de desempenho, mas só é necessário fazer este passo uma vez que o resultado é guardado e reutilizado posteriormente. Isto é obrigatório porque infelizmente o OpenCL não permite ao *host* conhecer o endereço de onde uma estrutura está alocada diretamente. A listagem 17 mostra o código usado para obter o endereço base.

4.4 Conclusão

A adição destas novas funcionalidades ao Marrow foi um processo necessário para eficientemente processar o grafo e a este serem aplicados algoritmos. Apesar destas funcionalidades serem adicionadas ao Marrow com o propósito de acelerar partes da construção do grafo ou na execução de algoritmos, a sua implementação foi generalizada para poder ser usada fora do contexto de grafos. Todas estas adições foram feitas a partir de funcionalidades base já existentes no Marrow. Isto facilitou os casos do *scan* e do filtro que foram implementados usando o modelo de programação em *skeletons* (já implementado no Marrow), o que integra automaticamente estas novas funcionalidades com as restantes que o Marrow já dispõe. A lista também usa componentes base para a sua implementação pois os endereços são armazenados em vetores do Marrow, no entanto foi necessário adicionar mais funcionalidades para incorporar a lista ao Marrow. A utilização da lista em GPU obriga necessariamente a execução de um *kernel* antes que sejam descobertos os endereços base, sendo este comportamento único à lista. Nos *kernels* também foi preciso

implementar a conversão de um endereço para uma estrutura de dados.

5

Avaliação

Neste capítulo é avaliado o desempenho da solução desenvolvida nesta tese, o Marrow-Graph, em comparação com outras bibliotecas de processamento de grafos na execução de diferentes algoritmos em vários grafos.

5.1 Objetivos

Nesta secção é avaliada a solução do Marrow-Graph, de modo a validar o seu objetivo como uma biblioteca de processamento de grafos. Para isso no Marrow-Graph são testados os tempos de execução de dois algoritmos, o Breadth-First Search (BFS) e Single Source Shortest Path (SSSP), contra outras bibliotecas do estado da arte em processamento de grafos em GPU: Gunrock e Hornet. Também é avaliado o impacto do tamanho das *slotted pages* na construção de alguns grafos.

Com esta avaliação queremos responder às seguintes questões:

- Há um tamanho ideal para uma *slotted page*?
- Como é que o Marrow-Graph se compara em relação ao Gunrock, no processamento de grafos dinâmicos?
- Como é que o Marrow-Graph se compara em relação ao Hornet, no processamento de grafos dinâmicos?

5.2 Metodologia

Para avaliar se há um tamanho ideal para uma *slotted page* é cronometrado o tempo de criação dos grafos listados na secção 5.3 em diferentes tamanho de *slotted pages*.

Na comparação do Marrow-Graph com o Gunrock e o Hornet, os mesmos grafos da secção 5.3 foram aplicados ao BFS e ao SSSP e os seus tempos foram cronometrados. Noutro teste, visto que estas soluções não suportam alterações dinâmicas ao grafo, são repetidos os testes tendo em conta o custo de reconstrução das estruturas de dados e

Nome do Grafo	Número de Vértices	Número de Arestas
ak2010	45.292	108.549
asia_osm	11.950.757	12.711.603
belgium_osm	1.441.295	1.549.970
coAuthorsDBLP	299.067	977.676
delaunay_n13	8.192	24.547
delaunay_n21	2.097.152	6.291.408
hollywood-2009	1.139.905	57.515.616
roadNet-CA	1.971.281	2.766.607
road_usa	23.947.347	28.854.312

Tabela 5.1: Diferentes grafos usados nos diferentes testes

GPU	CPU	RAM	Disco
NVIDIA GTX 970 (4GB GDDR5)	Intel i7-4770 @3.40 Ghz	16GiB DDR3 @1600 MHz	HDD: 1TB
NVIDIA Quadro M2000 (4GB GDDR5)	2x Intel Xeon E5-2609 v4 @1.7 Ghz	32GiB DDR4 @2400 MHz	SSD: 110GB

Tabela 5.2: Máquinas usadas para executar as experiências

o custo de transmissão entre *host* e dispositivo, comparando-os apenas com o custo de adição e transmissão das *slotted pages* alteradas no Marrow-Graph.

Em todas as avaliações, os testes foram efetuados 10 vezes, registando-se a média dos valores. Quando executados os algoritmos BFS e SSSP o vértice fonte escolhido foi o 0 em todos os grafos.

5.3 Grafos Considerados

A tabela 5.1 apresenta os diferentes grafos considerados para todos os testes efetuados. Estes grafos estão disponíveis no seguinte repositório `https://bitbucket.org/marrow-project/graphs` e foram escolhidos de modo a representar vários rácios entre o número de vértices e número de arestas. Todos estes grafos foram construídos a partir de dados reais.

5.4 O Ambiente de Testes

Hardware utilizado A tabela 5.2 apresenta as características do hardware utilizado no processo de avaliação e tem um fator limitante: a memória disponível para a GPU de

Nome do Grafo	Nº Vértices + Nº Arestas
delaunay_n13	32.739
ak2010	153.841
coAuthorsDBLP	1.276.743
belgium_osm	2.991.265
roadNet-CA	4.737.888
delaunay_n21	8.388.560
asia_osm	24.662.360
road_usa	52.801.659
hollywood-2009	58.655.521

Tabela 5.3: Tamanho dos grafos, obtido por a soma do número de vértices com o número de arestas

4GB. No entanto todos os grafos escolhidos na secção 5.3 estão abaixo deste limite para o Marrow-Graph, Gunrock e Hornet.

Para simplificar a distinção das máquinas, estas são referidas pela sua GPU, sendo a primeira máquina tratada por "GTX 970" e a segunda máquina por "M2000".

5.5 Resultados

5.5.1 Tamanho Ideal para uma *Slotted Page*

Estes testes foram efetuados de modo a tentar apurar qual o tamanho correto para as *slotted pages*, para isso foi cronometrado o tempo para construir todos os grafos da tabela 5.1 com diferentes tamanhos de páginas desde 256 a 2048 com um incremento para cada teste de 256. A tabela 5.3 mostra uma estimativa do tamanho do grafo ao somar o número de vértice com o número de arestas. Este teste apenas foi efetuado na máquina "GTX 970".

Como o tamanho dos grafos é drasticamente diferente são usadas percentagens de modo a comparar o efeito do tamanho das *slotted pages* no tempo de processamento dos grafos. No gráfico da figura 5.1, 100% significa que o grafo demorou o tempo máximo de todos os testes. As outras percentagens indicam o melhoramento face ao pior tempo. Por exemplo, uma percentagem de 89% indica que houve uma redução de 11% e que o tempo atual é 89% do pior tempo.

O gráfico 5.1 revela algumas tendências importantes, os 3 maiores grafos (asia_osm, road_usa e hollywood-2009) têm o pior desempenho quando a *slotted page* tem o tamanho de 256. Os 3 melhoram drasticamente os tempos quando o tamanho da página duplica e melhoram até o tamanho de 1280. Depois desse valor os tempos vão piorando marginalmente.

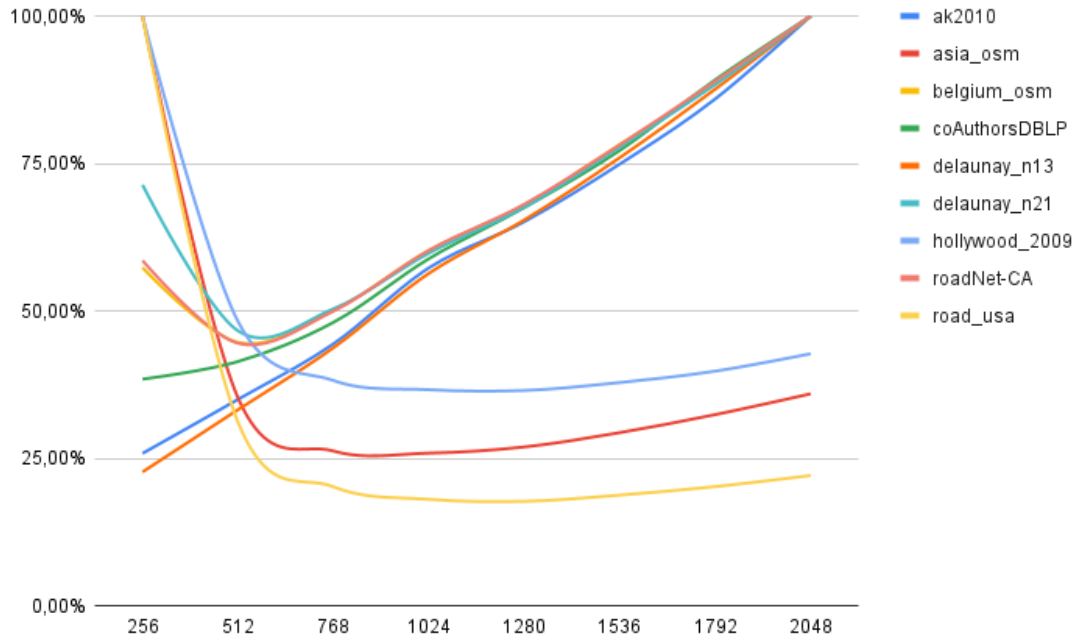

Figura 5.1: Gráfico que relaciona o tempo de construção do grafo com o tamanho das *slotted pages*

Os 3 grafos mais pequenos (delaunay_n13, ak2010 e coAuthorsDBLP) têm o melhor desempenho quando a *slotted page* tem o tamanho de 256 e o tempo piora à medida que o tamanho da *slotted page* aumenta.

Os 3 grafos restantes (belgium_osm, roadNet-CA e delaunay_n21), que podem ser designados como intermédios, têm o melhor desempenho quando a *slotted page* tem o tamanho de 512. Para tamanhos maiores que 512 os tempos pioram a um ritmo semelhante ao grupo dos grafos mais pequenos.

5.6 Comparação com o estado da arte

5.6.1 BFS - Breadth-first search

O gráfico da figura 5.2 compara os tempos de execução do BFS no Hornet e no Marrow-Graph em relação ao Gunrock, nos grafos descrito na secção 5.3, na máquina M2000. No gráfico são apresentados os valores do Hornet e do Marrow-Graph por cima das barras, sendo os valores do Gunrock omitidos por serem sempre igual a 1.

Valores menores que 1 indicam que o tempo foi inferior ao Gunrock, enquanto valores superiores a 1 indicam quantas vezes mais lento foi o tempo. Os valores dos tempos podem ser observados na tabela 5.4.

Grafo	Gunrock	Hornet	Marrow-Graph
delaunay_n13	4,53	4,10	38,88
ak2010	5,40	5,11	59,54
coAuthorsDBLP	5,25	4,73	206,70
belgium_osm	123,43	117,76	1647,73
roadNet-CA	52,74	49,41	999,59
delaunay_n21	56,76	57,32	1681,974
road_usa	682,43	639,12	12966,41
asia_osm	2970,71	2778,6	34227,15
hollywood-2009	87,26	71,88	8279,04

Tabela 5.4: Tempos de execução em milissegundos do algoritmo BFS na máquina M2000

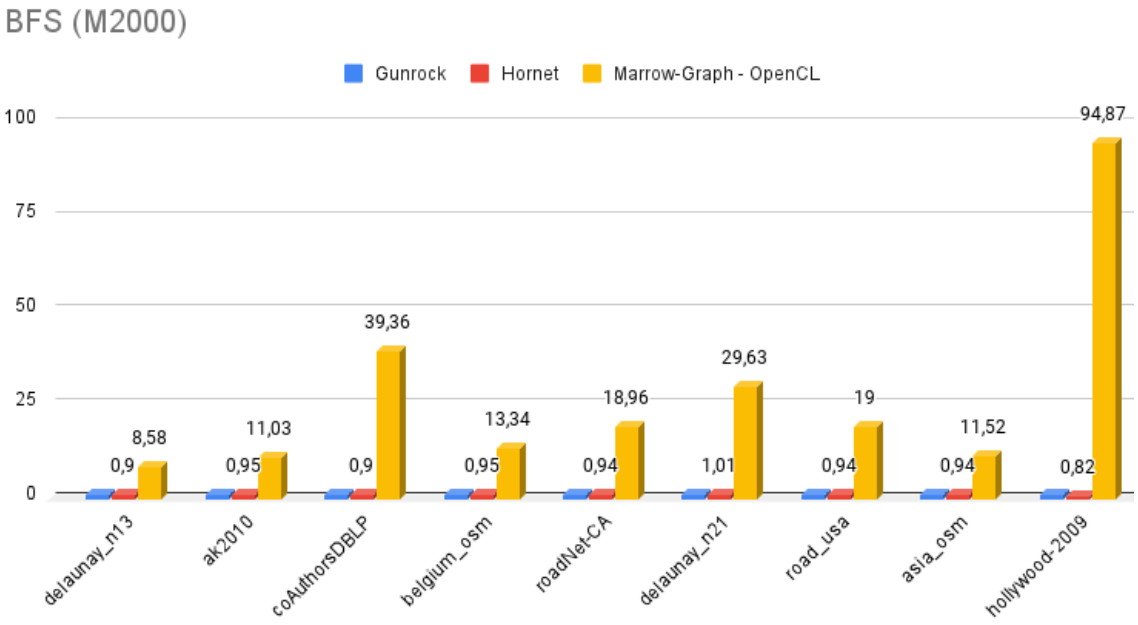

Figura 5.2: Comparação dos tempos execução do BFS na máquina M2000, relativos ao Gunrock.

A mesma experiência foi realizada para a máquina "GTX 970". Os resultados comparativos são apresentados na figura 5.3 e os respectivos valores absolutos presentes na tabela 5.5.

Observando ambos os gráficos é possível concluir que o Gunrock e o Hornet têm os tempos muito semelhantes, sendo em todos os testes o Marrow-Graph mais lento.

Nos dois gráficos há dois grafos que apresentam um desempenho consideravelmente mais alto que os restantes: coAuthorsDBLP e hollywood-2009 e não foi apurada a razão desta discrepância.

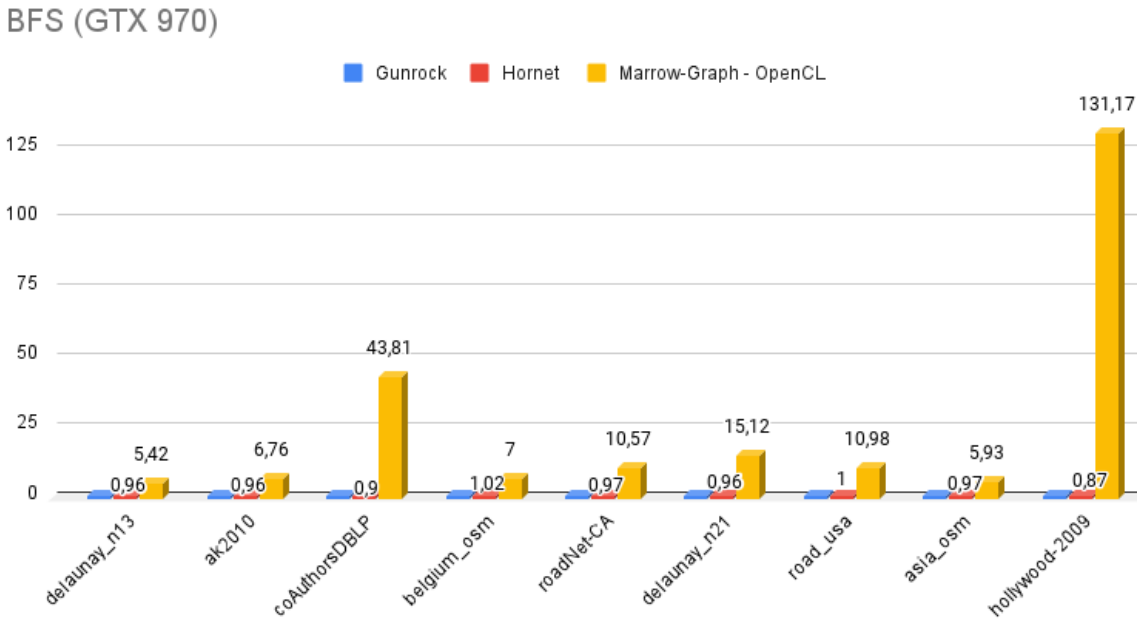

Figura 5.3: Comparação dos tempos execução do BFS na máquina GTX 970, relativos ao Gunrock.

Grafo	Gunrock	Hornet	Marrow-Graph
delaunay_n13	3,87	3,73	20,99
ak2010	4,72	4,52	31,96
coAuthorsDBLP	2,35	2,11	102,76
belgium_osm	115,39	117,22	807,9
roadNet-CA	48,29	46,82	510,4
delaunay_n21	50,3	48,3	760,72
road_usa	561,35	559,06	6163,99
asia_osm	3005,55	2905,27	17809,17
hollywood-2009	30,29	26,42	3973,615

Tabela 5.5: Tempos de execução em milissegundos do algoritmo BFS na máquina GTX 970

Comparando os valores entre as tabelas é aparente que os tempos em geral são mais baixos na máquina "GTX 970", especialmente em relação ao Marrow-Graph. Isto provavelmente deve-se à diferença de processadores entre as duas máquinas, onde apesar da máquina "M2000" ter mais núcleos, estes são mais lentos comparativamente à máquina "GTX 970". O maior número de núcleos não influencia a implementação do BFS do Marrow-Graph, pois o código executado no CPU é apenas executado num núcleo, sendo mais influenciado pela rapidez dos núcleos do que a sua quantidade.

Os gráficos 5.4 e 5.5 têm em consideração a capacidade de fazer alterações ao grafo

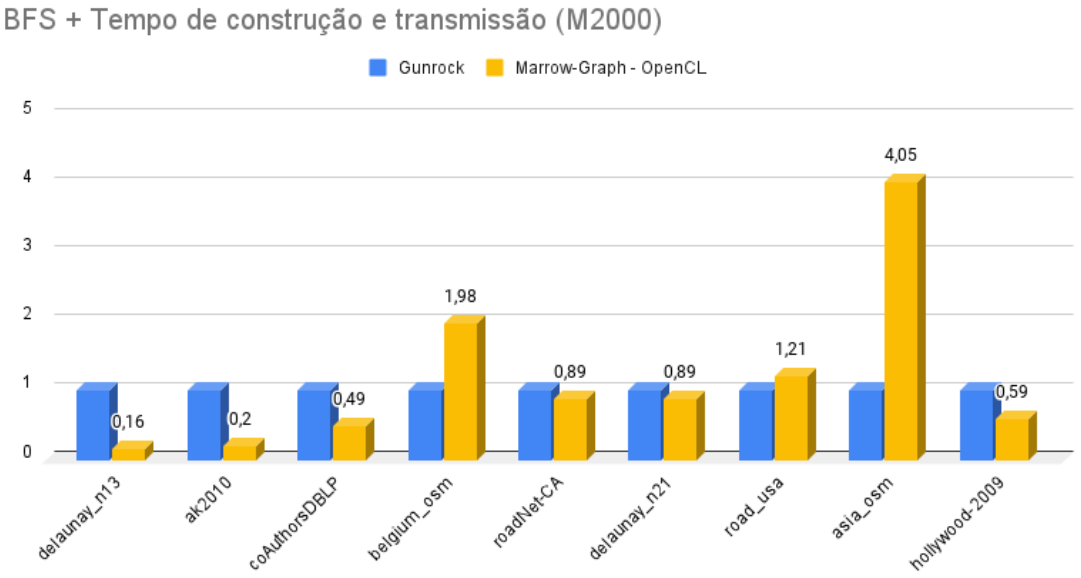

Figura 5.4: Diferença de tempos no algoritmo BFS mais o tempo de construção e transmissão na máquina M2000

Grafo	Gunrock	Marrow-Graph
delaunay_n13	244,41	39,163
ak2010	308,22	60,7
coAuthorsDBLP	439,64	217,29
belgium_osm	847,79	1678,49
roadNet-CA	1167,38	1038,03
delaunay_n21	1990,13	1775,37
road_usa	10832,46	13118,89
asia_osm	8489,37	34403,84
hollywood-2009	14146,48	8291

Tabela 5.6: Tempos de execução em milissegundos do algoritmo BFS mais o tempo de construção e transmissão na máquina M2000

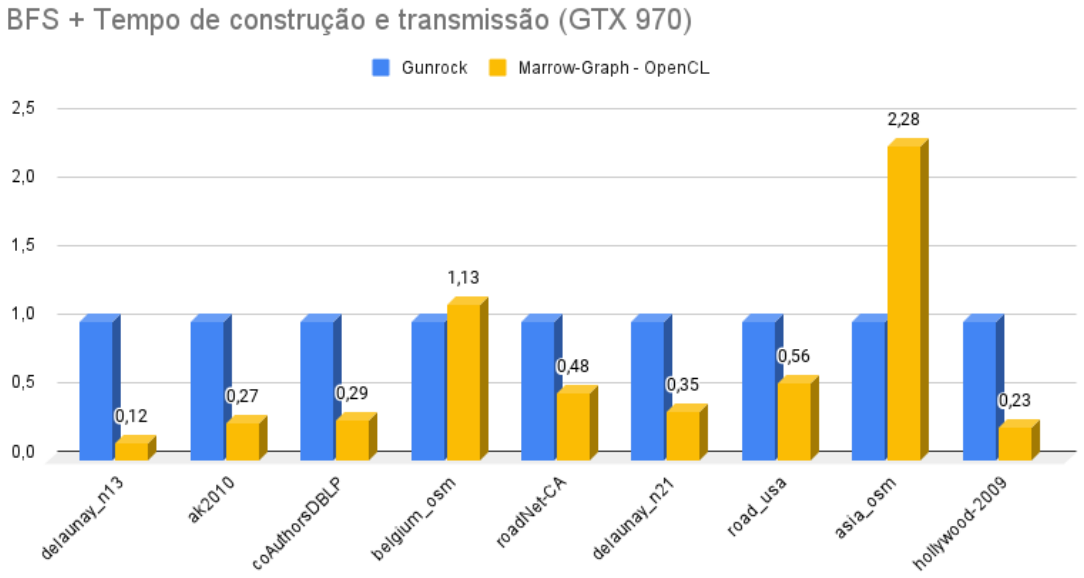

Figura 5.5: Diferença de tempos no algoritmo BFS mais o tempo de construção e transmissão na máquina GTX 970

Grafo	Gunrock	Marrow-Graph
delaunay_n13	175,25	21,2
ak2010	119,99	32,15
coAuthorsDBLP	355,54	103,33
belgium_osm	713,56	809,71
roadNet-CA	1057,7	512,9
delaunay_n21	2172,95	767,66
road_usa	11079,61	6179,43
asia_osm	7815,58	17812,7
hollywood-2009	16995,36	3980,62

Tabela 5.7: Tempos de execução em milissegundos do algoritmo BFS mais o tempo de construção e transmissão na máquina GTX 970

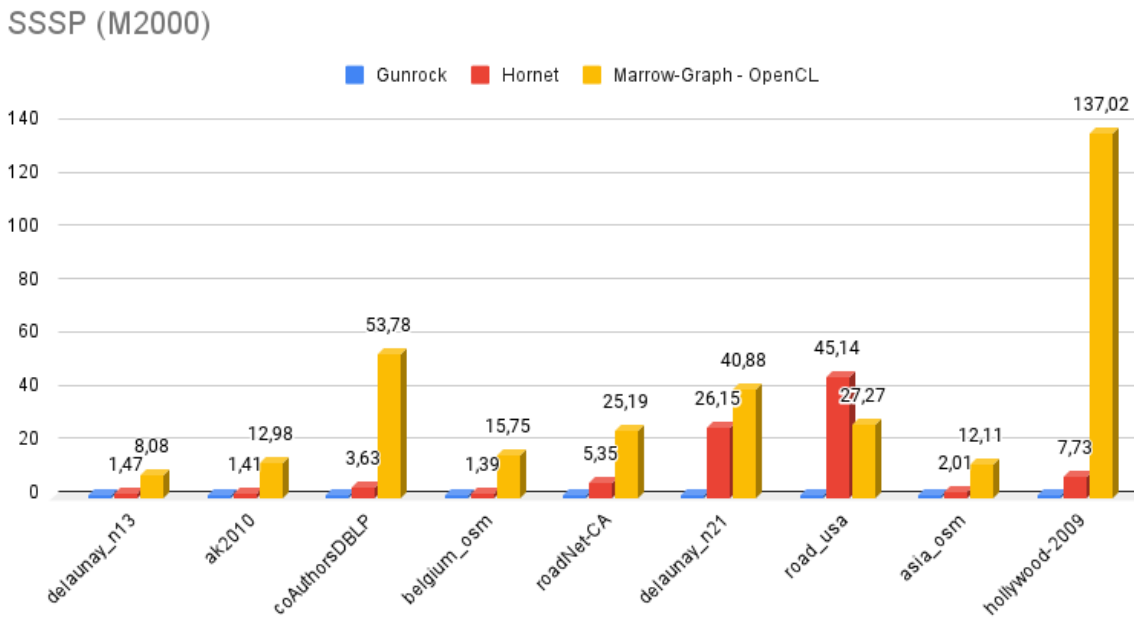

Figura 5.6: Comparação dos tempos execução do SSSP na máquina M2000, relativos ao Gunrock.

e contabiliza o custo de fazer duas adições de vértices unindo-os com uma aresta. Neste teste a comparação foi feita apenas com o Gunrock pois apenas este apresentava métricas detalhadas sobre o tempo de construção do grafo e a sua transmissão.

Comparando os tempos da máquina "M2000" o Marrow-Graph é mais rápido nos grafos: delaunay_n13, ak2010, coAuthorsDBLP, roadNet-CA, delaunay_n21 e hollywood-2009. Sendo o Gunrock mais rápido nos grafos: belgium_osm, road_usa e asia_osm.

O resultado da máquina "GTX 970" é semelhante ao da máquina "M2000", mas com a adição do grafo road_usa à lista de grafos que são processados em menos tempo no Marrow-Graph.

5.6.2 SSSP - Single source shortest path

O gráfico da figura 5.6 compara os tempos de execução do SSSP no Hornet e no Marrow-Graph em relação ao Gunrock, nos grafos descritos na secção 5.3, na máquina "M2000". No gráfico são apresentados os valores do Hornet e do Marrow-Graph por cima das barras, sendo os valores do Gunrock omitidos por serem sempre igual a 1. Os tempos em milissegundos podem ser observados na tabela 5.8.

O mesmo estilo de gráfico foi feito para a máquina "GTX 970" e está representado na figura 5.7 e os tempos de execução estão presentes na tabela 5.9.

Ao contrário do BFS, o Hornet tem tempos mais altos comparativamente ao Gunrock, tendo inclusive no grafo road_usa um tempo mais alto de execução que o Marrow-Graph.

Tal como o BFS o Marrow-Graph é aproximadamente 2 vezes mais rápido na máquina "GTX 970". Com a exceção do road_usa do Hornet, o Marrow-Graph é mais lento do que

Grafo	Gunrock	Hornet	Marrow-Graph
delaunay_n13	4,66	6,87	37,66
ak2010	5,87	7,9	72,5
coAuthorsDBLP	6,24	22,67	335,43
belgium_osm	125	173,65	1968,3
roadNet-CA	55,74	298,3	1404,04
delaunay_n21	62,08	1623,26	2538,06
road_usa	682,43	30804,68	18608,35
asia_osm	2948,55	5932,31	35710,185
hollywood-2009	103,62	800,77	14198,28

Tabela 5.8: Tempos de execução em milissegundos do algoritmo SSSP na máquina M2000

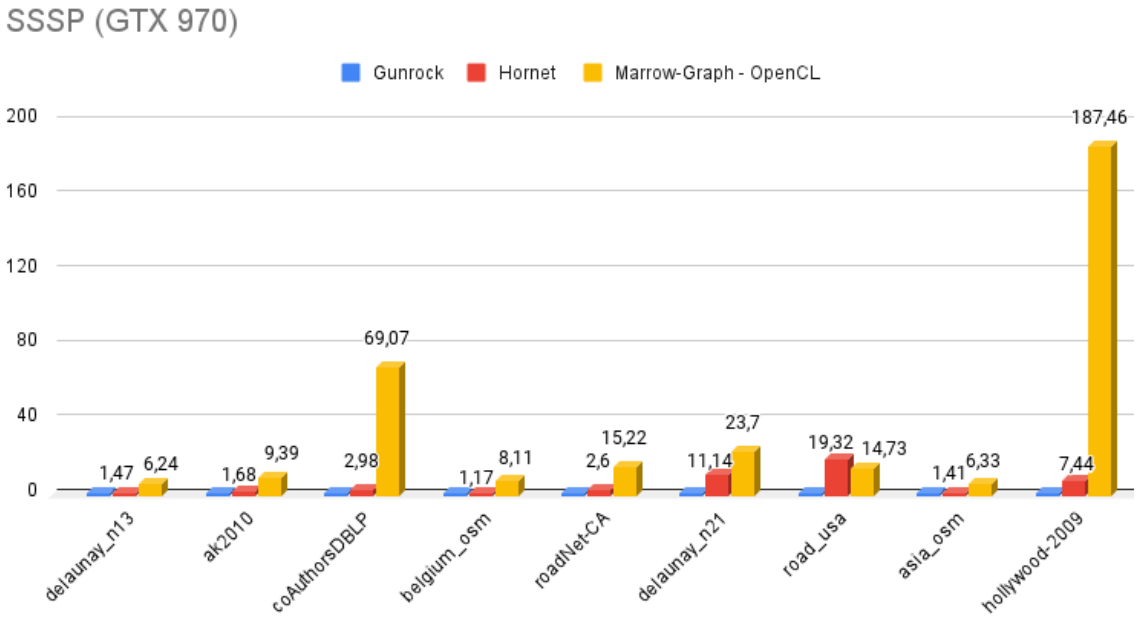

Figura 5.7: Comparação dos tempos execução do SSSP na máquina GTX 970, relativos ao Gunrock.

o Gunrock e o Hornet nos restante grafos.

Quando é avaliada a componente dinâmica do Marrow-Graph obtemos para a máquina "M200" o gráfico 5.8 e para a máquina "GTX 970" o gráfico 5.9. Na máquina "M2000", apenas os grafos delaunay_n13, ak2010, coAuthorsDBLP e hollywood-2009, foram mais rápidos do que o Gunrock. Na máquina "GTX 970" é obtido um resultado similar ao do teste feito com algoritmo BFS, onde os mesmos grafos: delaunay_n13, ak2010, coAuthorsDBLP, roadNet-CA, delaunay_n21, road_usa e hollywood-2009 foram mais rápidos comparativamente ao Gunrock.

Grafo	Gunrock	Hornet	Marrow-Graph
delaunay_n13	3,56	5,23	22,28
ak2010	4,09	6,87	38,416
coAuthorsDBLP	2,43	7,22	167,6
belgium_osm	113,21	132,45	918,47
roadNet-CA	48,28	125,69	734,862
delaunay_n21	50,74	565,42	1202,52
road_usa	580,65	11215,98	8555,72
asia_osm	2854,2	4011,43	18066,68
hollywood-2009	36,41	271,06	6825,87

Tabela 5.9: Tempos de execução em milissegundos do algoritmo SSSP na máquina GTX 970

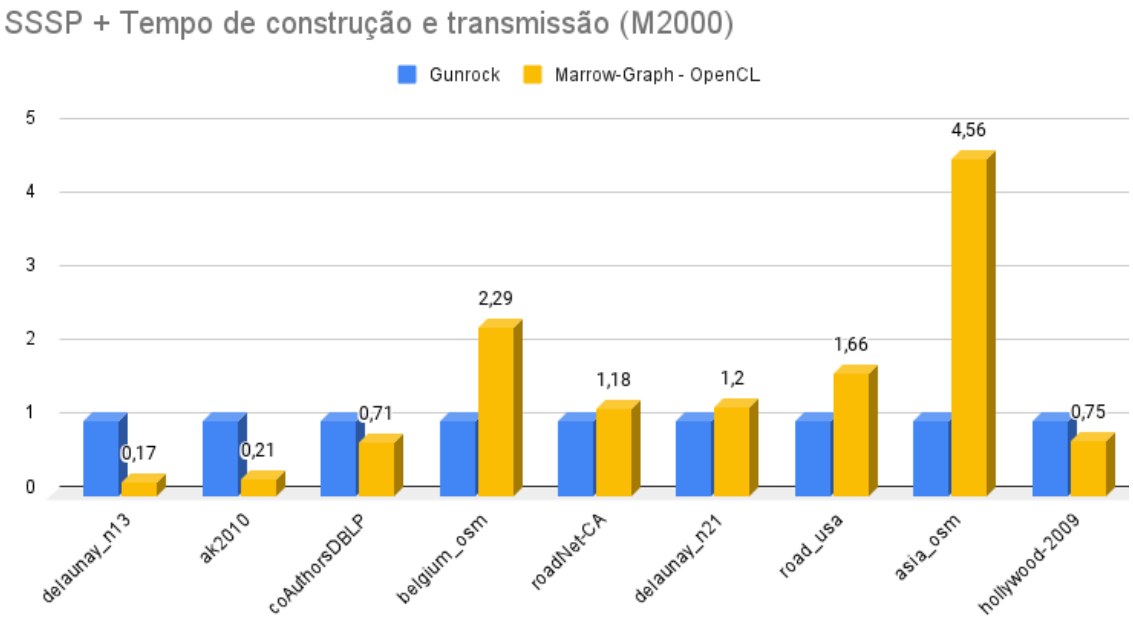

Figura 5.8: Comparação dos tempos execução do SSSP na máquina M2000, relativos ao Gunrock.

Os tempos de execução deste teste estão representados nas tabelas 5.10 e 5.11.

5.7 Conclusão

Com base nos testes efetuados é possível responder as perguntas colocadas na secção 5.1.

Há um tamanho ideal para uma *slotted page*? Não há um tamanho ideal que minimize a criação de qualquer grafo, parece haver uma relação entre o tamanho de um grafo,

Grafo	Gunrock	Marrow-Graph
delaunay_n13	219,76	37,94
ak2010	350,82	73,65
coAuthorsDBLP	489,9	346,02
belgium_osm	871,9	19990,05
roadNet-CA	1223,03	1442,21
delaunay_n21	2194,52	2631,46
road_usa	11320,56	18760,82
asia_osm	7864,28	35886,88
hollywood-2009	18877,02	14210,24

Tabela 5.10: Tempos de execução em milissegundos do algoritmo SSSP na máquina M2000

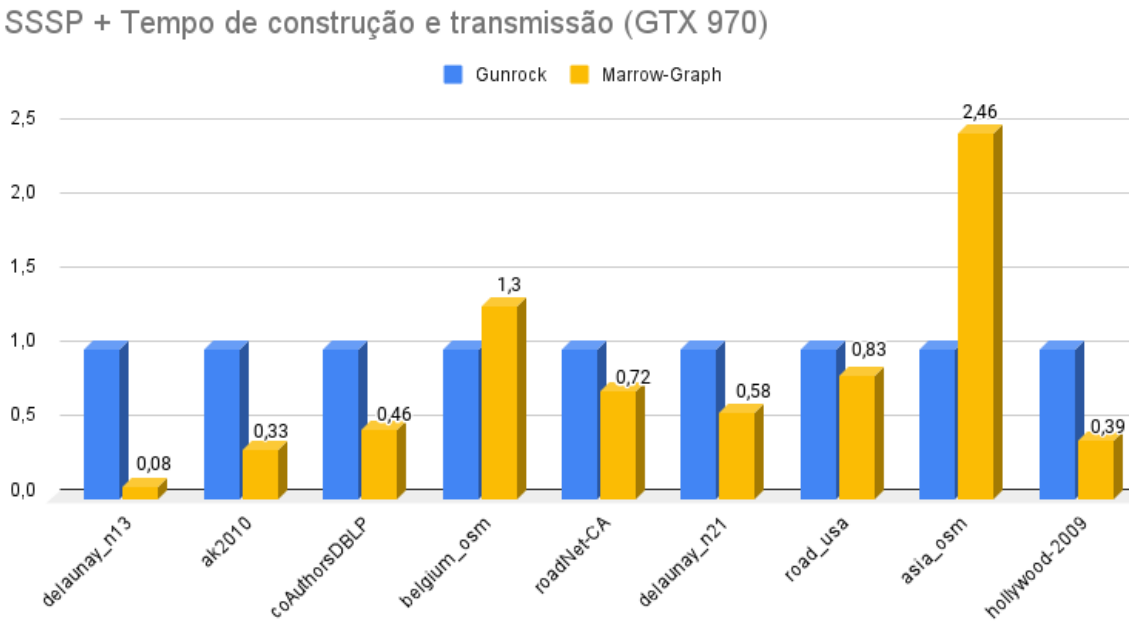

Figura 5.9: Comparação dos tempos execução do SSSP na máquina GTX 970, relativos ao Gunrock.

Grafo	Gunrock	Marrow-Graph
delaunay_n13	268,7	22,48
ak2010	117,67	38,64
coAuthorsDBLP	365,07	168,17
belgium_osm	708,23	919,28
roadNet-CA	1025,57	735,88
delaunay_n21	2079,43	1205,52
road_usa	10351,2	8562,37
asia_osm	7336,02	18070,33
hollywood-2009	17472,59	6834,64

Tabela 5.11: Tempos de execução em milissegundos do algoritmo SSSP na máquina GTX 970

obtido pela soma do número de vértice com o número de arestas, e o tamanho de uma *slotted page*. De acordo com os testes efetuados, quanto maior o tamanho do grafo, maior precisa de ser o tamanho de uma *slotted page*. É difícil de afirmar que esta conclusão é verdade para todos os grafos, mas mostra que é preciso ter alguma consideração a escolher o tamanho de uma *slotted page*.

Como é que o Marrow-Graph se compara em relação ao Gunrock e o Hornet? Inicialmente o Hornet foi escolhido para comparação com o Marrow-Graph por apresentar também um grafo dinâmico, só foi verificado mais tarde que não possuía uma implementação dinâmica para o BFS e o SSSP. O desempenho do Gunrock e do Hornet foi muito similar ao longo dos testes, por isso a conclusão aplica-se a ambos. O Marrow-Graph é mais lento na maioria os testes diretos com o Gunrock e o Hornet, no entanto, isto é expectável, pois há uma sobrecarga significativa no operador *advance* do Marrow-Graph para procurar em que página se encontram os vértices e onde se localizam em cada página, parte da diferença de tempos também se atribui ao facto que tanto o Gunrock como o Hornet utilizam CUDA ao contrário do Marrow-Graph que usa OpenCL, sendo o OpenCL tipicamente mais lento que o CUDA na execução do mesmo código.

Por outro lado, quando é avaliada a sua componente dinâmica quando comparada às soluções estáticas do Gunrock e Hornet, o Marrow-Graph torna-se mais apelativo. Ao dividir o grafo em diferentes *slotted pages* contribui para reduzir o tempo de transmissão de dados entre o dispositivo e o *host* no caso de alterações ao grafo. Estas poupanças de tempo são grandes o suficiente para compensar os tempos mais lentos do Marrow-Graph a executar os algoritmos testados. No entanto nem todos os grafos se revelaram mais rápidos, demonstrando que atualmente é preciso ter em consideração o grafo antes de escolher o Marrow-Graph.

6

Conclusões e Trabalho Futuro

6.1 Conclusões

Esta tese teve como objetivo principal o desenvolvimento de uma biblioteca para o processamento de grafos dinâmicos em GPU. O método escolhido para alcançar esse objetivo foi a utilização conjunta do Marrow com a própria representação do grafo em *slotted pages*. Desta forma obteve-se um grafo capaz de sofrer adições, edições e remoções, cujo estado em CPU e GPU é automaticamente gerido pelo Marrow. Com a separação do grafo em diferentes páginas a probabilidade de ser necessária uma retransmissão do grafo todo reduz significativamente. Deste modo foi possível desenvolver um grafo verdadeiramente dinâmico, capaz de sofrer alterações drásticas no seu tamanho no decorrer da sua execução.

Outro objetivo foi desenvolver uma biblioteca de processamento de grafos em GPU. Estas funcionalidades foram implementadas no Marrow-Graph, fazendo uso da *framework* do Marrow e o seu desempenho foi avaliado na secção 5 sendo implementado dois algoritmos usando essas mesmas funcionalidades. Há uma penalidade computacional em usar as *slotted pages* como estrutura para representar um grafo, essa penalidade foi clara durante a avaliação, quando a solução implementada nesta tese foi comparada com outras as soluções estáticas. No entanto quando são quantificados os benefícios de ter um grafo dinâmico em testes em que a construção do grafo e sua respetiva transmissão faz com que a solução do Marrow-Graph seja apelativa. Os testes revelam que o conceito funciona, no entanto, para ser considerada uma biblioteca de processamento de dados é necessário mais trabalho além do que foi concluído nesta tese.

O Marrow foi uma ferramenta extremamente poderosa ao longo desta tese, sendo que as funcionalidades que oferece atacaram logo desde início alguns problemas fulcrais no desenvolvimento de um grafo dinâmico e de uma biblioteca de processamento de grafos. Com esta tese, o Marrow demonstrou-se flexível de modo a acomodar novas funcionalidades ficando mais equipado para futuros projetos devolvidos enquanto *framework*.

6.2 Trabalho Futuro

Como trabalho futuro proponho:

1. Que a construção do grafo seja otimizada.

   A implementação atual não escala com o número de núcleos disponíveis, usando apenas um. Trazer uma implementação *multithreaded* iria beneficiar o Marrow-Graph e reduzir os tempos de execução.

2. Implementação de mais algoritmos.

   Por restrições de tempo apenas foram implementados dois algoritmos, o BFS e SSSP, no entanto seria interessante ver como se comporta o Marrow-Graph em mais algoritmos.

Bibliografia

[1] F. Alexandre, R. Marqués e H. Paulino. "On the support of task-parallel algorithmic skeletons for multi-GPU computing". Em: *Symposium on Applied Computing, SAC 2014, Gyeongju, Republic of Korea - March 24 - 28, 2014*. ACM, 2014, pp. 880–885. DOI: 10.1145/2554850.2555018 (ver p. 20).

[2] S. Ashkiani et al. "GPU LSM: A Dynamic Dictionary Data Structure for the GPU". Em: *CoRR* abs/1707.05354 (2017). arXiv: 1707.05354. URL: http://arxiv.org/abs/1707.05354 (ver p. 19).

[3] B. F. Auer e R. H. Bisseling. "A GPU Algorithm for Greedy Graph Matching". Em: *Facing the Multicore - Challenge II - Aspects of New Paradigms and Technologies in Parallel Computing [Proceedings of a conference held at the Karlsruhe Institute of Technology (KIT), September 28-30, 2011]*. Ed. por R. Keller, D. Kramer e J. Weiss. Vol. 7174. Lecture Notes in Computer Science. Springer, 2011, pp. 108–119. ISBN: 978-3-642-30396-8. DOI: 10.1007/978-3-642-30397-5_10. URL: https://doi.org/10.1007/978-3-642-30397-5_10 (ver p. 12).

[4] U. Banerjee et al., eds. *International Conference on Supercomputing, ICS'12, Venice, Italy, June 25-29, 2012*. ACM, 2012. ISBN: 978-1-4503-1316-2. URL: http://dl.acm.org/citation.cfm?id=2304576 (ver p. 8).

[5] G. E. Blelloch. *Vector Models for Data-Parallel Computing*. MIT Press, 1990. ISBN: 0-262-02313-X (ver p. 13).

[6] A. M. Caulfield et al. "A cloud-scale acceleration architecture". Em: *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE. 2016, pp. 1–13 (ver p. 2).

[7] A. Ching et al. "One Trillion Edges: Graph Processing at Facebook-Scale". Em: *Proc. VLDB Endow.* 8.12 (ago. de 2015), pp. 1804–1815. ISSN: 2150-8097. DOI: 10.14778/2824032.2824077. URL: https://doi.org/10.14778/2824032.2824077 (ver p. 1).

[8] A. Gharaibeh et al. "A yoke of oxen and a thousand chickens for heavy lifting graph processing". Em: *International Conference on Parallel Architectures and Compilation Techniques, PACT '12, Minneapolis, MN, USA - September 19 - 23, 2012*. Ed. por P. Yew et al. ACM, 2012, pp. 345–354. ISBN: 978-1-4503-1182-3. DOI: 10.1145/2370816.2370866. URL: http://doi.acm.org/10.1145/2370816.2370866 (ver p. 16).

[9] O. Green e D. A. Bader. "cuSTINGER: Supporting dynamic graph algorithms for GPUs". Em: *2016 IEEE High Performance Extreme Computing Conference, HPEC 2016, Waltham, MA, USA, September 13-15, 2016*. IEEE, 2016, pp. 1–6. ISBN: 978-1-5090-3525-0. DOI: 10.1109/HPEC.2016.7761622. URL: https://doi.org/10.1109/HPEC.2016.7761622 (ver p. 18).

[10] C.-Y. Gui et al. "A survey on graph processing accelerators: Challenges and opportunities". Em: *Journal of Computer Science and Technology* 34.2 (2019), pp. 339–371 (ver pp. 2, 16).

[11] W.-S. Han et al. "TurboGraph: a fast parallel graph engine handling billion-scale graphs in a single PC". en. Em: *Proceedings of the 19th ACM SIGKDD international conference on Knowledge discovery and data mining - KDD '13*. Chicago, Illinois, USA: ACM Press, 2013, p. 77. ISBN: 9781450321747. DOI: 10.1145/2487575.2487581. URL: http://dl.acm.org/citation.cfm?doid=2487575.2487581 (acedido em 19/09/2019) (ver pp. 13, 28).

[12] P. Harish e P. J. Narayanan. "Accelerating Large Graph Algorithms on the GPU Using CUDA". Em: *High Performance Computing - HiPC 2007, 14th International Conference, Goa, India, December 18-21, 2007, Proceedings*. Ed. por S. Aluru et al. Vol. 4873. Lecture Notes in Computer Science. Springer, 2007, pp. 197–208. ISBN: 978-3-540-77219-4. DOI: 10.1007/978-3-540-77220-0_21. URL: https://doi.org/10.1007/978-3-540-77220-0_21 (ver p. 12).

[13] M. Harris. *How to Overlap Data Transfers in CUDA C/C++*. https://devblogs.nvidia.com/how-overlap-data-transfers-cuda-cc/. Visitado a 07 Jul 2018. 2012 (ver p. 10).

[14] M. Harris, S. Sengupta e J. D. Owens. *Chapter 39. Parallel Prefix Sum (Scan) with CUDA*. https://developer.nvidia.com/gpugems/gpugems3/part-vi-gpu-computing/chapter-39-parallel-prefix-sum-scan-cuda. Visitado a 20 Jul 2019 (ver pp. 47, 48, 50, 52).

[15] V. Hindriksen. *Comparing Syntax for CUDA, OpenCL and HiP*. https://streamhpc.com/blog/2016-04-05/comparing-syntax-cuda-opencl-hip/. Visitado a 07 Jul 2018. 2016 (ver p. 9).

[16] J. Hoberock e D. Tarjan. *Introduction to Massively Parallel Computing*. https://github.com/jaredhoberock/stanford-cs193g-sp2010/blob/master/lectures/lecture_1/introduction_to_massively_parallel_computing.pdf. Visitado a 07 Jul 2018. 2010 (ver p. 5).

[17] S. T. Kelly e M. A. Black. "graphsim: An R package for simulating gene expression data from graph structures of biological pathways". Em: *Journal of Open Source Software* 5.51 (2020), p. 2161. DOI: 10.21105/joss.02161. URL: https://doi.org/10.21105/joss.02161 (ver p. 1).

[18] J. Kim et al. "SnuCL: an OpenCL framework for heterogeneous CPU/GPU clusters". Em: *International Conference on Supercomputing, ICS'12, Venice, Italy, June 25-29, 2012*. Ed. por U. Banerjee et al. ACM, 2012, pp. 341–352. ISBN: 978-1-4503-1316-2. DOI: 10.1145/2304576.2304623. URL: https://doi.org/10.1145/2304576.2304623 (ver p. 8).

[19] G. Malewicz et al. "Pregel: a system for large-scale graph processing". Em: *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2010, Indianapolis, Indiana, USA, June 6-10, 2010*. Ed. por A. K. Elmagarmid e D. Agrawal. ACM, 2010, pp. 135–146. ISBN: 978-1-4503-0032-2. DOI: 10.1145/1807167.1807184. URL: http://doi.acm.org/10.1145/1807167.1807184 (ver p. 16).

[20] R. Marqués et al. "Algorithmic Skeleton Framework for the Orchestration of GPU Computations". Em: *Euro-Par 2013 Parallel Processing - 19th International Conference, Aachen, Germany, August 26-30, 2013. Proceedings*. Vol. 8097. Lecture Notes in Computer Science. Springer, 2013, pp. 874–885. DOI: 10.1007/978-3-642-40047-6_86 (ver pp. 3, 20).

[21] D. Merrill, M. Garland e A. S. Grimshaw. "Scalable GPU graph traversal". Em: *Proceedings of the 17th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPOPP 2012, New Orleans, LA, USA, February 25-29, 2012*. Ed. por J. Ramanujam e P. Sadayappan. ACM, 2012, pp. 117–128. ISBN: 978-1-4503-1160-1. DOI: 10.1145/2145816.2145832. URL: http://doi.acm.org/10.1145/2145816.2145832 (ver pp. 12, 13).

[22] P. Micikevicius. *Fundamental Optimizations*. http://www.nvidia.com/content/PDF/sc_2010/CUDA_Tutorial/SC10_Fundamental_Optimizations.pdf. Visitado a 07 Jul 2018. 2010 (ver p. 11).

[23] NVIDIA Corporation. *CUDA C Best Practices Guide*. https://docs.nvidia.com/cuda/cuda-c-best-practices-guide/index.html. Visitado a 07 Jul 2018. 2018 (ver p. 10).

[24] NVIDIA Corporation. *GPU Gems 2*. Visitado a 09 Jul 2018. 2005 (ver pp. 19, 20).

[25] NVIDIA Corporation. *NVIDIA Tesla P100. The Most Advanced Datacenter Accelerator Ever Built*. Featuring Pascal GP100, the World's Fastest GPU. Rel. téc. Visitado a 07 Jul 2018. 2017 (ver p. 6).

[26] J. van Oosten. *CUDA Thread Execution Model*. https://www.3dgep.com/cuda-thread-execution-model/. Visitado a 07 Jul 2018. 2011 (ver pp. 8, 9, 11).

[27] J. D. Owens et al. "A survey of general-purpose computation on graphics hardware". Em: *Computer graphics forum*. Vol. 26. 1. Wiley Online Library. 2007, pp. 80–113 (ver p. 2).

[28] S. Sahu et al. "The Ubiquity of Large Graphs and Surprising Challenges of Graph Processing". Em: *Proc. VLDB Endow.* 11.4 (dez. de 2017), pp. 420–431. ISSN: 2150-8097. DOI: 10.1145/3164135.3164139. URL: https://doi.org/10.1145/3164135.3164139 (ver p. 1).

[29] M. Sha et al. "Accelerating Dynamic Graph Analytics on GPUs". Em: *PVLDB* 11.1 (2017), pp. 107–120. URL: http://www.vldb.org/pvldb/vol11/p107-sha.pdf (ver p. 1).

[30] X. Shi et al. "Graph Processing on GPUs: A Survey". Em: *ACM Computing Surveys (CSUR)* 50.6 (2018), p. 81 (ver pp. 2, 7, 8, 12–14).

[31] P. Singal e R. Chhillar. "Dijkstra shortest path algorithm using global positioning system". Em: *International Journal of Computer Applications* 101.6 (2014), pp. 12–18 (ver p. 1).

[32] F. Soldado, F. Alexandre e H. Paulino. "Execution of compound multi-kernel OpenCL computations in multi-CPU/multi-GPU environments". Em: *Concurrency and Computation: Practice and Experience* 28.3 (2016), pp. 768–787. DOI: 10.1002/cpe.3612. URL: https://doi.org/10.1002/cpe.3612%7D (ver p. 20).

[33] J. Soman, K. Kothapalli e P. J. Narayanan. "A fast GPU algorithm for graph connectivity". Em: *24th IEEE International Symposium on Parallel and Distributed Processing, IPDPS 2010, Atlanta, Georgia, USA, 19-23 April 2010 - Workshop Proceedings*. IEEE, 2010, pp. 1–8. DOI: 10.1109/IPDPSW.2010.5470817. URL: https://doi.org/10.1109/IPDPSW.2010.5470817 (ver p. 12).

[34] Y. Wang et al. "Gunrock: GPU Graph Analytics". Em: *CoRR* abs/1701.01170 (2017). arXiv: 1701.01170. URL: http://arxiv.org/abs/1701.01170 (ver p. 16).

[35] Wikipedia, the free encyclopedia. *Illustration of a BSP superstep*. https://en.wikipedia.org/wiki/File:Bsp.wiki.fig1.svg. Visitado a 09 Jul 2018. 2006 (ver p. 17).

[36] C. Zeller. *CUDA C/C++ Basics. Supercomputing 2011 Tutorial*. https://www.nvidia.com/docs/IO/116711/sc11-cuda-c-basics.pdf. Visitado a 07 Jul 2018. 2011 (ver p. 6).

[37] J. Zhong e B. He. "Medusa: Simplified Graph Processing on GPUs". Em: *IEEE Trans. Parallel Distrib. Syst.* 25.6 (2014), pp. 1543–1552. DOI: 10.1109/TPDS.2013.111. URL: https://doi.org/10.1109/TPDS.2013.111 (ver p. 16).