UNIVERSIDADE FEDERAL DE GOIÁS
INSTITUTO DE INFORMÁTICA

MATEUS FERREIRA E FREITAS

Projeto e Avaliação de Algoritmos Paralelos para Sistemas Multicore e Manycore Aplicados no Processamento de Documentos

Goiânia
2017

sistema de bibliotecas ufg

PRPG

PRÓ-REITORIA DE PÓS-GRADUAÇÃO

UFG

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

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

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

2. Identificação da Tese ou Dissertação:

Nome completo do autor: Mateus Ferreira e Freitas

Título do trabalho: Projeto e Avaliação de Algoritmos Paralelos para Sistemas Multicore e Manycore Aplicados no Processamento de Documentos

3. Informações de acesso ao documento:

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

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

Mateus Ferreira e Freitas

Assinatura do(a) autor(a)[2]

Ciente e de acordo:

Assinatura do(a) orientador(a)[2]

Data: 29 / 09 / 2017

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

Casos de embargo:
- Solicitação de registro de patente
- Submissão de artigo em revista científica
- Publicação como capítulo de livro
- Publicação da dissertação/tese em livro

[2] A assinatura deve ser escaneada.

Mateus Ferreira e Freitas

Projeto e Avaliação de Algoritmos Paralelos para Sistemas Multicore e Manycore Aplicados no Processamento de Documentos

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

Área de concentração: Ciência da Computação.

Orientador: Prof. Wellington Santos Martins

Goiânia
2017

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

Freitas, Mateus Ferreira e
Projeto e Avaliação de Algoritmos Paralelos para Sistemas Multicore e Manycore Aplicados no Processamento de Documentos [manuscrito] / Mateus Ferreira e Freitas. - 2017.
0 97 f.: il.

Orientador: Prof. Dr. Wellington Santos Martins.
Dissertação (Mestrado) - Universidade Federal de Goiás, Instituto de Informática (INF), Programa de Pós-Graduação em Ciência da Computação, Goiânia, 2017.
Bibliografia.
Inclui abreviaturas, tabelas, algoritmos, lista de figuras, lista de tabelas.

1. Paralelismo. 2. GPU. 3. Aprendizado ativo. 4. Learning to rank. 5. Busca top-k. I. Martins, Wellington Santos, orient. II. Título.

CDU 004

MINISTÉRIO DA EDUCAÇÃO
UNIVERSIDADE FEDERAL DE GOIÁS
INSTITUTO DE INFORMÁTICA
PROGRAMA DE PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO

ATA Nº 24/2017

ATA DA SESSÃO DE JULGAMENTO DA DISSERTAÇÃO DE MESTRADO DE MATEUS FERREIRA E FREITAS

Aos trinta dias do mês de agosto de dois mil e dezessete, às catorze horas, na sala 150 do Instituto de Informática da Universidade Federal de Goiás, Campus Samambaia, reuniu-se a banca examinadora designada na forma regimental pela Coordenação do Curso para julgar a dissertação de mestrado intitulada **"Projeto e Avaliação de Algoritmos Paralelos para Sistemas Multicore e Manycore Aplicados no Processamento de Documentos"**, apresentada pelo aluno Mateus Ferreira e Freitas como parte dos requisitos necessários à obtenção do grau de Mestre em Ciência da Computação, área de concentração Ciência da Computação. A banca examinadora foi presidida pelo orientador do trabalho de dissertação, Professor Doutor Wellington Santos Martins (INF/UFG), tendo como membros os Professores Doutores Leonardo Andrade Ribeiro (INF/UFG) e Alba Cristina Magalhães de Melo (CIC/UnB). Aberta a sessão, o candidato expôs seu trabalho. Em seguida, o aluno foi arguido pelos membros da banca e:

(X) tendo demonstrado suficiência de conhecimento e capacidade de sistematização do tema de sua dissertação, a banca concluiu pela **aprovação** do candidato, sem restrições.

(   ) não tendo demonstrado suficiência de conhecimento e capacidade de sistematização do tema de sua dissertação, a banca concluiu pela **reprovação** do candidato.

Os trabalhos foram encerrados às 16:00 horas. Nos termos do Regulamento Geral dos Cursos de Pós-Graduação desta Universidade, lavrou-se a presente ata que, lida e julgada conforme, segue assinada pelos membros da banca examinadora.

Prof. Dr. Wellington Santos Martins ______________________

Prof. Dr. Leonardo Andrade Ribeiro ______________________

Profa. Dra. Alba Cristina Magalhães de Melo ______________________

Todos os direitos reservados. É proibida a reprodução total ou parcial do trabalho sem autorização da universidade, do autor e do orientador(a).

Mateus Ferreira e Freitas

Graduou-se em Ciência da Computação na UFG - Universidade Federal de Goiás. Durante sua graduação, foi monitor de Tópicos Avançados de Programação no Instituto de Informática da UFG. Durante o Mestrado foi bolsista da CAPES, e desenvolveu soluções paralelas de aprendizado de máquina usando placas gráficas no grupo de pesquisa LDA - Laboratório de Computação de Alto Desempenho e Aplicações.

Dedico este trabalho à minha família.

Agradecimentos

Agradeço minha família: Mãe Geralda da Penha Ferreira, Pai Fausto Ribeiro de Freitas e Irmão Fausto Ribeiro de Freitas Júnior, por terem me apoiado e ajudado durante o desenvolvimento deste trabalho.

Agradeço ao meu orientador Wellington Santos Martins, por me guiar se disponibilizar para orientação, tirando minhas dúvidas e me dando um ótimo suporte neste trabalho.

Agradeço ao Daniel Xavier de Sousa e Thierson Couto Rosa, por suas colaborações e sugestões.

"*Machine learning is the next Internet.*"

Tony Tether, Director of DARPA,

.

Resumo

e Freitas, Mateus Ferreira. **Projeto e Avaliação de Algoritmos Paralelos para Sistemas Multicore e Manycore Aplicados no Processamento de Documentos** . Goiânia, 2017. 97p. Dissertação de Mestrado. Instituto de Informática, Universidade Federal de Goiás.

Diversas aplicações processam documentos de diferentes maneiras, visando filtrá-los, organizá-los ou aprender com eles. Atualmente, é necessário um grande poder computacional para que isso seja feito eficientemente, devido ao número grande e crescente de documentos. Geralmente os documentos são independentes entre si, o que facilita o uso de paralelismo para acelerar esse processamento. Este trabalho explora três problemas: aprendizado ativo, *learning to rank* (L2R) e busca top-k. Usando o paralelismo em CPUs *multicore* e GPUs (*Graphics Processing Unit*) *manycore*, algoritmos paralelos foram propostos e avaliados para cada problema, e implementados com as APIs OpenMP e CUDA. Para problema de aprendizado ativo foi proposto um algoritmo *multicore*, que obteve *speedup* de 10,8x no melhor caso com 12 *threads*. A versão *manycore* proposta obteve *speedup* de 128x em relação ao serial, e uma solução com 4 GPUs atingiu 3,5x de *speedup* sobre 1 GPU. Para o problema de L2R foi proposto um algoritmo *manycore*, que segue uma abordagem por bloco de *threads* usando o conceito de *Combinadic*, e usa uma *cache* com *fingerprint* para acelerar o processamento. Os *speedups* nos melhores casos foram de 508x sobre o serial, 9x sobre uma *baseline* em GPU, e 4x sobre nossa solução com 1 GPU ao usar 4 GPUs. Ao comparar com uma versão sem o *combinadic*, o *speedup* sobre ela foi de 4,4x com ambas versões usando 1 GPU e 3,9x usando 4. Estas soluções usaram estruturas de mapa de *bits* para acelerar a criação de regras de associação. Na busca top-k foram implementadas uma solução serial e uma *multicore* de um algoritmo *manycore* estado da arte para buscas exatas. Estas implementações serviram de *baseline* para nossa extensão desse algoritmo, que inclui o uso de multi-GPU, buscas em grupos e um balanceamento de carga intra-bloco. Os *speedups* obtidos foram de 2,7x sobre o algoritmo original, 17x sobre o serial, 4x sobre o *multicore*, e 4x sobre nossa versão ao usar 4 GPUs.

Palavras–chave

Paralelismo, GPU, Regras de Associação, Learning to Rank, Aprendizado Ativo, Busca Top-K

Abstract

e Freitas, Mateus Ferreira. . Goiânia, 2017. 97p. MSc. Dissertation. Instituto de Informática, Universidade Federal de Goiás.

Several applications process documents in different ways, aiming to filter, organize or learn with them. Nowadays, a great computational power is necessary in order to do that efficiently, due to the large and increasing number of documents. Usually, documents are independent of each other, which facilitates the use of parallelism to speed up this processing. This work explores three problems: active learning, learning to rank (L2R) and top-k search. Using the parallelism on multicore CPUs and manycore GPUs (Graphics Processing Unit), parallel algorithms were proposed and evaluated for each problem, and implemented with the OpenMP and CUDA APIs.

For the active learning problem a multicore algorithm was proposed, which obtained 10.8x of speedup in the best case with 12 threads. The proposed manycore version obtained 128x of speedup over the serial version, and a solution with 4 GPUs achieved 3.5x of speedup over 1 GPU.

For the L2R problem a manycore algorithm was proposed, which follows a thread-block approach using the concept of Combinadic, and uses a cache with fingerprint to speed up the processing. The best case speedups were 508x over the serial, 9x over a GPU baseline, and 4x over our solution when using 4 GPUs. When comparing with a version without combinadic, the speedup over it was 4.4x with both versions using 1 GPU and 3.9x with 4. These solutions used bitmap structures to speed up the association rules creation.

In the top-k search a serial and multicore solutions were implemented from a state of the art manycore algorithm for exact searches. These implementations served as baselines for our extension of this algorithm, which includes the use of multi-GPU, group searches and an intra-block load balancing. The speedups were 2.7x over the original algorithm, 17x over the serial, 4x over the multicore, and 4x over our version when using 4 GPUs.

Keywords

Parallelism, GPU, Association Rules, Learning to Rank, Active learning, Top-K Search

Sumário

Lista de Figuras **13**

Lista de Tabelas **14**

Lista de Algoritmos **15**

Lista de Códigos de Programas **16**

1 Introdução **17**
- 1.1 Objetivo e Contribuições 20
- 1.2 Estrutura do Trabalho 20

2 Arquitetura CUDA **21**
- 2.1 Implementação do Hardware 21
- 2.2 Kernel e Hierarquia de Threads 23
- 2.3 Hierarquia de Memória 24
- 2.4 Arquitetura Kepler 26
- 2.5 Redução em CUDA 31

3 Fundamentação Teórica e Trabalhos Relacionados **35**
- 3.1 Itemsets Frequentes e Regras de Associação 35
  - 3.1.1 Representação do Banco de Dados e Itemsets 35
  - 3.1.2 Mineração de Itemset e de Regras de Associação 37
  - 3.1.3 Outros Trabalhos Relacionados 42
- 3.2 Aprendizado Ativo 43
  - 3.2.1 Aprendizado Ativo com Regras de Associação 43
  - 3.2.2 Outros Trabalhos Relacionados 45
- 3.3 Learning to Rank 46
  - 3.3.1 Learning to Rank com Regras de Associação 47
  - 3.3.2 Parallel Learning to Rank with Association Rules 49
  - 3.3.3 Outros Trabalhos Relacionados 50
- 3.4 k-Vizinhos Mais Próximos 51
  - 3.4.1 Indexação dos Dados 52
  - 3.4.2 Busca dos k-Vizinhos Mais Próximos 53
  - 3.4.3 Algoritmo GPU-based Textual k-Nearest Neighbors (GT-kNN) 55
  - 3.4.4 Outros Trabalhos Relacionados 55

4 Algoritmos e Implementações Propostas **57**
4.1 Parallel SSARP (PSSARP) 57
4.1.1 Os Algoritmos PSSAR e PSSARP 57
4.1.2 Ilustração do PSSAR 58
4.2 Parallel Rule-based Selective Sampling (PRSS) 59
4.2.1 Representação dos Dados e Abordagem por Thread 59
4.2.2 O Algoritmo PRSS 59
4.2.3 PRSS com Múltiplas GPUs 61
4.3 Parallel Rule-based On the Fly Learning to Rank (PROFL) 61
4.3.1 Abordagem por Bloco de Threads 62
4.3.2 Combinadic 63
4.3.3 Reaproveitando Itemsets 63
4.3.4 O Algoritmo PROFL 64
4.3.5 PROFL com Múltiplas GPUs 65
4.3.6 Ilustração do PROFL 66
4.4 MultiThreaded k-Nearest Neighbors (MT-kNN) 66
4.4.1 Busca na CPU 67
4.4.2 O Algoritmo MT-kNN 67
4.5 Fine-grained Sparse and High-dimensional k-Nearest Neighbors (FiSH-kNN) 68
4.5.1 Balanceamento de Carga por Bloco de Threads 69
4.5.2 Processamento de Grupo de Consultas 69
4.5.3 O Algoritmo FiSH-kNN 71
4.5.4 FiSH-kNN com Múltiplas GPUs 71

5 Experimentos e Resultados **73**
5.1 Configuração da Máquina Utilizada 73
5.2 Resultados do PSSARP e PRSS 73
5.2.1 Características das Reduções 74
5.2.2 Tempo de Processamento das Reduções 74
5.3 Resultados do PROFL 76
5.3.1 Tempo de Processamento do PROFL 77
5.3.2 Resultados do PROFL com abordagem por *threads* 77
5.3.3 Análise do Uso da Cache 80
5.4 Resultados do MT-kNN e FiSH-kNN 83
5.4.1 Tempo de Processamento de uma Consulta 84
5.4.2 Análise do Uso de Grupos de Consultas 87
5.4.3 Análise do Tipo de Balanceamento 88

6 Conclusão **90**
6.1 Trabalhos Futuros 91
6.2 Artigos Publicados 92

Referências Bibliográficas **93**

Lista de Figuras

1.1 Visão geral de uma máquina de busca. Adaptado de Manning et al. [22]. 18

2.1 Escalabilidade automática [26]. 22
2.2 Hierarquia de *threads* [26]. 24
2.3 Hierarquia da memória [26]. 26
2.4 Escalonador de *warp* na arquitetura Kepler [24]. 28
2.5 Hierarquia de memória da arquitetura Kepler [24]. 28
2.6 Exemplos de acesso na memória global [26]. 30
2.7 Execução sequencial de kernel e execução com concorrência [26]. 31
2.8 Sobreposição de transferência (Fonte: Mark Harris)[15]. 31
2.9 Exemplo de redução por soma de um vetor usando CUDA [25]. 34

3.1 a) Banco de dados binário. b) Banco de dados horizontal. c) Banco de dados vertical [42]. 37
3.2 *Itemsets* frequentes da Figura 3.1 [42]. 38
3.3 Grade de *itemsets*, onde o traço em negrito representa a árvore de prefixo [42]. 39
3.4 Visão geral do SSAR. 44
3.5 Visão geral do SSARP. 45
3.6 Exemplo de interseção de mapas de *bits*, quando o número de documentos é 16 [10]. 50
3.7 Criação do índice invertido [7]. 53
3.8 Processamento de uma consulta contendo três termos [7]. 54

4.1 Visão geral das implementações *multicore* e *manycore* do PSSAR. 58
4.2 Ilustração do PROFL. 67
4.3 Ilustração do PROFL sem o *combinadic*. 68
4.4 Processamento de um grupo de três consultas. 71

Lista de Tabelas

2.1 Recursos computacionais da capacidade computacional 3.5. 27

5.1 Médias das características de cada *fold* dos conjuntos de dados. 74
5.2 Médias das características de cada *fold* dos conjuntos de dados reduzidos. 75
5.3 Tempo de execução em segundos das implementações SSARP e PRSS. 75
5.4 Speedup do PRSS com múltiplas GPUs usando regras de tamanho 4. 76
5.5 Tempo de execução em segundos das implementações SSARP e PSSARP. 76
5.6 Tempo de execução em segundos do PROFL. Melhores resultados estão em negrito. 78
5.7 Speedup do PROFL com múltiplas GPUs sobre o LRAR e PLRAR, usando regras de tamanho 4. 78
5.8 Speedup do PROFL com múltiplas GPUs sobre uma GPU, usando regras de tamanho 4. 79
5.9 Tempo de execução em segundos do PROFL com abordagem por *threads*. 80
5.10 Speedup do PROFL_T com múltiplas GPUs usando regras de tamanho 4. 81
5.11 Speedup do PROFL sobre o PROFL com abordagem por *threads*. 82
5.12 Quantidade de posições disponíveis na cache. 82
5.13 Cache hit e miss do PROFL. 83
5.14 Informação geral dos *datasets* textuais. 84
5.15 Tempo em milissegundos para uma consulta. 85
5.16 Tempo em segundos da indexação. 85
5.17 Speedup do FiSH-kNN em relação aos outros. 86
5.18 Speedup do FiSH-kNN ao usar múltiplas GPUs. 86
5.19 Speedup sequencial em relação ao rainbow, e com multithreading. 86
5.20 Tempo em segundos do processamento da entrada dos métodos kNN. 87
5.21 Speedup da indexação do FiSH-kNN sobre os outros. 87
5.22 Pico de memória em Megabytes. 87
5.23 Tempo em milissegundos para uma consulta com variados tamanhos de grupo. 88
5.24 Speedup do uso de grupos. 88
5.25 Tempo de consulta com balanceamento intra-bloco e *speedup* sobre o balanceamento inter-bloco. 88
5.26 Speedup do balanceamento inter-bloco sobre o intra-bloco, com uso de grupos. 89

Lista de Algoritmos

1 BruteForce - Algoritmo força bruta de mineração de *itemsets* frequentes [42]. 40
2 AssociationRules - Algoritmo de mineração de regras de associação [42]. 41
3 SSAR - Selective Sampling using Association Rules. 44
4 SSARP - Selective Sampling using Association Rules with Partitions. 44
5 LRAR - Learning to Rank with Association Rules. 48
6 PLRAR - Parallel Learning to Rank with Association Rules. 50
7 DataIndexing($\mathcal{E}$) 53
8 kNNSearch(invertedIndex, q) 55
9 GPU-based Textual-kNN($\mathcal{E}$, $\mathcal{Q}$) 55
10 PSSAR - Parallel Selective Sampling using Association Rules. 58
11 PSSARP - Parallel Selective Sampling using Association Rules w/ Partitions. 58
12 PRSS - Parallel Rule-based Selective Sampling. 60
13 Rule Counter. 61
14 Multi-GPU PRSS. 62
15 Combinadic – Geração de um subconjunto de valores inteiros. 63
16 PROFL – Parallel Rule-based On the Fly Learning to Rank. 65
17 Multi-GPU PROFL. 66
18 CPU-kNNSearch(invertedIndex, q) 68
19 MultiThreaded-kNN($\mathcal{E}$, $\mathcal{Q}$) 69
20 Block Balanced kNNSearch(invertedIndex, q) 70
21 FiSH-kNNSearch(invertedIndex, $\mathcal{G}$, $\mathcal{B}$) 70
22 Fine-grained Sparse and High-dimensional-kNN($\mathcal{E}$, $\mathcal{Q}$, $\mathcal{B}$) 71
23 Multi-GPU FiSH-kNN($\mathcal{E}$, $\mathcal{Q}$, $\mathcal{B}$, #GPUs) 72

Lista de Códigos de Programas

2.1 Exemplo de *kernel* [26]. 23
2.2 void reduce6() [16] 33

CAPÍTULO 1

Introdução

Um dos maiores fluxos de dados na Web hoje em dia são os resultados de pesquisas. As páginas web podem ser consideradas como documentos e serem tratadas como tal. Há um grande interesse em aprender com esses documentos, e é preciso grande poder computacional para isso já que há uma quantidade crescente e enorme de documentos. Também é necessário atender requisições de usuários com eficiência, o que é dificultado por essa quantidade de documentos.

Uma das aplicações sob documentos Web é o *Learning to Rank*, onde o objetivo é ordenar o resultado da pesquisa web, de forma que seja relevante para o usuário que fez a consulta [22]. Os trabalhos [6, 9, 39, 41, 40, 37, 18] focam apenas em criar um único modelo, que não permite a flexibilidade [8] de se trabalhar sob demanda. Isso devido aos seus métodos de minimização de erros, e/ou maximização de métricas, que necessitam ser aplicados várias vezes na fase de treinamento, para gerar um modelo complexo de alta acurácia [8]. O LRAR (*Learning to Rank with Association Rules*) [38] é uma solução que tem esse aspecto sob demanda ao utilizar regras de associação [2], e que possui oportunidades para aprimorar com o uso de paralelismo.

Muitas vezes os documentos Web são ruidosos, pouco informativos, ou redundantes, e para isso existe o aprendizado ativo. O aprendizado ativo cria uma coleção reduzida de documentos, fazendo uma seleção de acordo com algum critério [23, 20], mas sempre buscando melhorar ou manter a qualidade dos modelos de *ranking* que serão criados usando essa versão reduzida. A solução SSARP (*Selective Sampling using Association Rules with Partitions*) [33] é utilizada em outros trabalhos como [34, 10], e consegue criar uma coleção pequena, informativa e diversa da coleção original. A nosso conhecimento, essa solução nunca foi paralelizada, e uma versão paralela permitiria trabalhar com *datasets* de milhares de consultas nunca antes processados. Além disso permitirá que os trabalhos relacionados utilizassem essa nova solução mais eficiente.

O algoritmo de kNN (*k-Nearest Neighbors*) possui várias aplicações, como geração de *metafeatures* [7], classificação [30], cálculo de similaridade [44], sistemas de recomendação [1], filtragem colaborativa [29], entre vários outros. O trabalho [7] possui uma versão estado da arte para kNN textual (GT-kNN GPU-based Textual k-NN), um

problema de alta dimensionalidade e esparsidade. Sua versão trabalha somente com uma única consulta por vez, e não utiliza múltiplas GPUs. Otimizações nessa versão estado da arte melhoraria diversas aplicações de kNN, principalmente as que necessitam executar o kNN múltiplas vezes, como a geração de *metafeatures* e filtragem colaborativa.

A Figura 1.1 ilustra de modo geral uma máquina de busca [22], que seria uma aplicação que utiliza estes três problemas. Uma coleção de documentos é indexada, para quando um usuário fazer uma consulta, apenas os Top-k similares são buscados. Os documentos dessa busca serão ordenados de maneira que os mais relevantes para a consulta estejam no início da lista. Isso é feito aplicando um modelo de ranqueamento sob esses documentos. O modelo é criado a partir de uma base de treino que contém documentos rotulados com variados níveis de relevância em relação às consultas armazenadas no treino. Essa base de treino é obtida por meio de aprendizado ativo aplicado sobre uma base não rotulada, onde especialistas são consultados para dar uma relevância para cada amostra selecionada. Também pode ser aplicado na base de treino para reduzir amostras ruidosas ou redundantes, para permitir que o modelo de ranqueamento seja criado mais rapidamente, e melhore ou pelo menos mantenha a efetividade de ranqueamento.

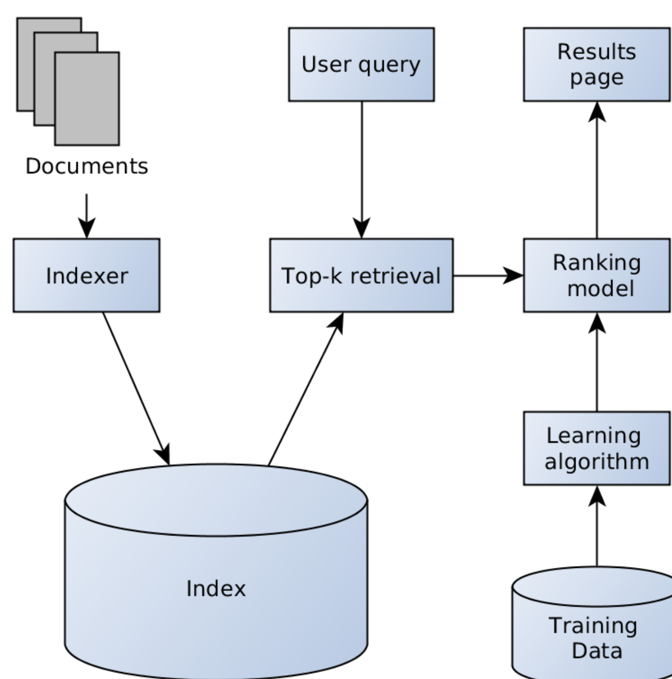

Figura 1.1: *Visão geral de uma máquina de busca. Adaptado de Manning et al. [22].*

Considerando que geralmente os documentos são independentes entre si, o processamento paralelo desses documentos é facilitado, e nos motivou a criar soluções paralelas *multicore* (CPU) e *manycore* (GPU) para esses três problemas. Para o método SSARP propomos a versão *multicore* PSSARP (*Parallel SSARP*), que obteve um speedup de 10,8x no melhor caso, e usando 12 *threads* de CPU. Além disso, propomos a versão

manycore PRSS (*Parallel Rule-based Selective Sampling*), onde a implementação com 1 GPU obteve 128x de speedup em relação ao SSARP. Uma implementação multi-GPU obteve 3,5x de speedup sobre o PRSS, ao usar 4 GPUs.

O PROFL (*Parallel Rule-based On the Fly Learning to Rank*) é nossa proposta *manycore* do LRAR, que explora mais oportunidades de paralelismo que uma versão *manycore* do LRAR, o PLRAR [10] (*Parallel LRAR*). O PROFL segue uma abordagem por bloco de *threads* usando o conceito de Combinadic, que gera um elemento de um conjunto de combinações lexicográfica dado um índice inteiro [5]. Juntamente com o PRSS, foram utilizadas estruturas de mapas de bits vetorizadas para acelerar a criação de regras de associação. Para aproveitar processamentos passados, também foi implementado uma *cache* com uso de *fingerprint* [4] para identificar unicamente as entradas em uma tabela *hash* e acelerar o processamento. Também foi implementado o uso de multi-GPU, e nossos experimentos utilizaram as bases reduzias geradas pelo PRSS. Os speedups nos melhores casos foram de 508x sobre o LRAR, 9x sobre o PLRAR, e 4x sobre o PROFL com 1 GPU ao usar 4 GPUs. Para comparação, uma versão com abordagem por *threads* foi implementada, onde o PROFL com abordagem por blocos obteve um speedup de 4,4x com ambos usando 1 GPU, e 3,9x com ambos usando 4 GPUs. Estas implementações (PRSS e PROFL) permitem a redução de bases com milhares de consultas e com variados parâmetros, e viabilizam o aspecto sob demanda do LRAR com nossa proposta paralela PROFL.

Para o kNN, foram propostas atualizações e avaliações do GT-kNN, para que possa ser aplicado com mais eficiência nas tarefas que precisam de executar o kNN várias vezes. Por ser aplicado em bases textuais, esse método é apropriado apenas para bases com alta esparsidade e alta dimensionalidade. A fim de tornar o GT-kNN em uma solução mais geral de busca Top-K e avaliá-lo, criamos nossa proposta FiSH-kNN (*Fine-grained Sparse and High-dimensional - k Nearest Neighbors*). No FiSH-kNN foi implementado o uso de multi-GPU, buscas em grupos de consultas, e um balanceamento de carga intra-bloco. Nossos experimentos mostraram que o balanceamento de carga intra-bloco difere no máximo em 4% do inter-bloco, e que o uso de grupos de consulta permanece em torno de 10% de ganho de speedup com grupos de tamanho 5 ou mais. Para comparação, também implementamos o FiSH-kNN serial e *multicore*, chamado MT-kNN (*MultiThreaded kNN*). Os speedups de melhor caso obtidos foram de 2,7x sobre o GT-kNN, 17x sobre o MT-kNN sequencial e 4x sobre o MT-kNN com 12 *threads*, 65x sobre a ferramenta sequencial Rainbow, 29x sobre o G-kNN [30] (*GPU-based kNN*), e 4x sobre o FiSH-kNN com 1 gpu ao usar 4 GPUs.

Nossa implementações foram feitas utilizando as APIs OpenMP para *multicore* e CUDA para *manycore*. Nas próximas seções descrevemos nossos objetivos, contribuições e a estrutura do trabalho.

1.1 Objetivo e Contribuições

O objetivo deste trabalho é desenvolver soluções paralelas para plataformas *multicore* e *manycore*, focando no processamento de documentos. Essas soluções darão a possibilidade de se trabalhar com bases de dados maiores com milhares de consultas.

Este trabalho possui as seguintes contribuições:

- Algoritmos paralelos *multicore* e *manycore* do método de aprendizado ativo SSARP e uma implementação multi-GPU, que viabilizam a aplicação em bases com milhares de consultas e com variadas parâmetros.
- Um algoritmo paralelo *manycore* do método de *learning to rank* LRAR, com uma abordagem por bloco de threads, uso de *cache* com *fingerprint* e implementação multi-GPU, que viabilizou a aplicação em bases com milhares de consultas.
- Extensões do método GT-kNN que incluem: implementação multi-GPU, buscas em grupos, e implementação de um balanço de carga intra-bloco. A avaliação de cada um foi feita.
- Implementação serial e *multicore* da versão estendida do GT-kNN.

1.2 Estrutura do Trabalho

- No 2º Capítulo são mostrados a arquitetura CUDA e seu o modelo de programação *manycore*.
- No 3º Capítulo está a fundamentação teórica e trabalhos relacionados, onde é explicado sobre mineração de *itemsets* frequentes e regras de associação; *learning to rank* ; aprendizado ativo; e busca Top-k com vizinhos mais próximos.
- No 4º Capítulo descrevemos nossos algoritmos e implementações propostos, o PSSARP, PRSS, PROFL, MT-kNN e FiSH-kNN.
- No 5º Capítulo há os experimentos e resultados de cada implementação proposta, aplicadas em bases de dados padronizadas.
- O 6º e último Capítulo contém nossa conclusão, sugestões para trabalhos futuros, e artigo publicado.

CAPÍTULO 2

Arquitetura CUDA

A GPU (*Graphics Processing Unit*) hoje em dia é um processador *manycore* com alto poder computacional devido ao seu elevado paralelismo. Seu poder computacional está sempre crescendo, sendo impulsionado principalmente pela área de animação 3D de alta definição, em jogos e filmes, por exemplo, e em aplicações de aprendizado de máquina. A GPU tem seu potencial totalmente explorado quando é utilizada para fazer computações com paralelismo de dados, que é quando o mesmo código é executado sobre vários dados em paralelo. Então para ter mais usos dessa plataforma, a arquitetura CUDA (*Compute Unified Device Architecture*) foi introduzida pela NVIDIA como uma plataforma de processamento paralelo de propósito geral. Descrevemos a seguir a arquitetura usando os guias da NVIDIA [26] e [24] (Arquitetura Kepler) como referências.

A GPU CUDA tem multiprocessadores de fluxo (*Streaming Multiprocessor* - SM), onde são executados blocos de *threads* da aplicação paralela, e cada bloco é escalonado para qualquer multiprocessador disponível. Isso permite uma escalabilidade automática, em que uma GPU com mais SMs executará mais rapidamente do que uma com menos SMs, como podemos ver na Figura 2.1.

2.1 Implementação do Hardware

Durante a invocação de um *kernel*, os blocos de seu *grid* são enumerados e distribuídos para serem executados em multiprocessadores *multithreaded* escaláveis. Quando o número de blocos excede a capacidade dos multiprocessadores, os blocos excedentes são executados assim que um multiprocessador está disponível. Os SMs encontram-se dispostos em vetores na arquitetura CUDA. Como um bloco pode conter centenas de *threads*, o SM precisa executar essa quantidade concorrentemente. De modo a gerenciar essa grande quantidade de *threads*, é empregada uma arquitetura única chamada SIMT (*Single–Instruction*, *Multiple Thread*). Essa arquitetura combina o SIMD (*Single–Instruction*, *Multiple Data*) com o *multithreading*, onde várias *threads* compartilham uma instrução e a aplica sobre múltiplos dados.

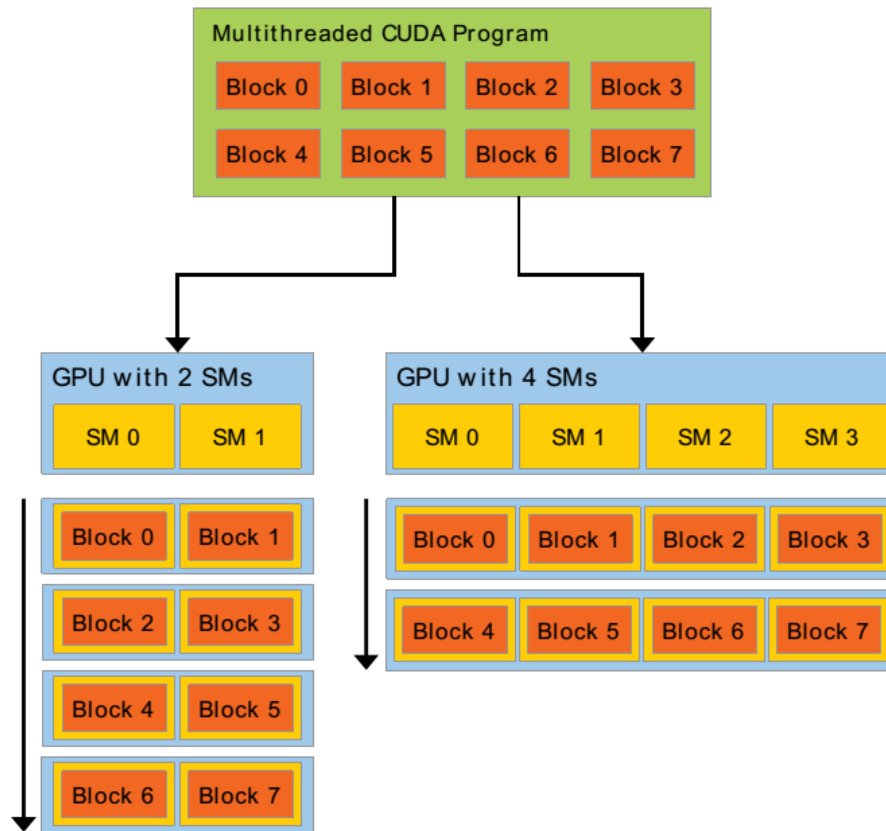

Figura 2.1: *Escalabilidade automática [26].*

O SM cria, gerencia, escalona, e executa *threads* em grupos de 32 *threads* paralelas chamadas de *warps*. Todas as *threads* de uma *warp* começam na mesma instrução do código, porém elas podem divergir em possíveis ramificações desse código, como em estruturas condicionais. Essa divergência é possível pois cada *thread* possui seu próprio estado de registro.

Warps são executadas nos processadores CUDA *core* que compõem o SM. Quando um ou mais blocos de *threads* são atribuídos a um SM, são particionados em *warps* e cada *warp* é escalonada por um escalonador de *warps* para executarem. O particionamento ocorre sempre do mesmo modo, com a primeira *warp* contendo a *thread* 0 e as restantes com IDs sequenciais.

Durante sua execução, a *warp* executa uma instrução comum entre suas *threads* por vez. Assim, a eficiência máxima só é atingida se todas 32 *threads* não divergirem no código. Quando divergem, as *threads* que divergiram são executadas primeiro e as restantes são desativadas, e depois convergem novamente para a mesma instrução quando todas outras divergentes terminam. Cada *warp* tem seu contexto de execução armazenado dentro do chip do SM durante toda sua execução. Assim, a mudança de contexto de execução não tem custos.

O que determina a capacidade do SM de executar *warps* e blocos de *threads* em

paralelo, para um dado *kernel*, depende dos recursos computacionais do dispositivo: o número de registradores 32 *bits* particionados entre as *warps* e a memória compartilhada entre os blocos de *threads*. Essas limitações, entre outras, são descritas pela capacidade computacional da GPU, que veremos a na Seção 2.4.

2.2 Kernel e Hierarquia de Threads

O CUDA estende a linguagem C, permitindo que funções chamadas *kernels* possam ser definidas. Estas funções, quando chamadas no código, são executadas N vezes em paralelo por N diferentes *threads* da plataforma CUDA.

Um *kernel* é definido com o especificador `__global__`, e para escolher o número de blocos e de *threads* em cada bloco que irão executar este *kernel*, deve-se usar um especificador de configuração `<<<...>>>` antes da chamada. Para funções normais, que poderão ser chamadas dentro do *kernel*, o especificador `__device__` é utilizado. Cada *thread* é identificada por um ID único que lhe é dado, que é acessível pela variável interna *threadIdx*. O código 2.1 exemplifica a soma paralela de dois vetores A e B de tamanho N, com o resultado armazenado em outro vetor C [26].

Código 2.1 Exemplo de *kernel* [26].

```
1  __device__ float sum(float a, float b){
2    return a + b;
3  }
4  __global__ void VecAdd(float* A, float* B, float* C){
5    int i = threadIdx.x;
6    C[i] = sum(A[i] , B[i]);
7  }
8  int main(){
9  ...
10   // Kernel com N threads e 1 bloco de threads
11   VecAdd<<<1, N>>>(A, B, C);
12 ...
13 }
```

As *threads* são aglomeradas em blocos de *threads*, e ambos podem ser dispostos em até 3 dimensões, sendo acessados por `threadIdx.x`, `.y` ou `.z` e `blockIdx` para o bloco. Também podemos acessar a quantidade de *threads* em cada dimensão de um bloco pela variável interna `blockDim`. Veja que no código 2.1 o especificador do *kernel* foi configurado com 1 bloco, e N *threads*. Como as *threads* são agrupados em blocos, elas são executadas em um mesmo SM, e assim, compartilham os recursos disponíveis. O que

implica que existe um limite máximo de *threads* para não estourar esses recursos. Este limite atualmente é de 1024 *threads* por bloco. Mas isso não quer dizer que podemos ter apenas 1024 execuções paralelas em CUDA. Um *kernel* pode ser executado por mais de um bloco, com todos tendo a mesma configuração, o que faz o total de *threads* ser o número de blocos vezes o número de *threads* em um bloco.

Os blocos são organizados em uma estrutura chamada *grid* de blocos como podemos ver na Figura 2.2 uma configuração de 2x3 blocos de 3x4 *threads* (`kernel<<< (2,3), (3,4) >>>(...)`). Os blocos dispõem de uma memória compartilhada, o que permite que as *threads* que nele residem façam um processamento cooperativo. Para isso é necessário uma sincronização na sua execução para coordenar os acessos à memória. Em CUDA C há uma função que faz sincronização por barreira - `__syncthreads()` -, onde todas as *threads* do bloco irão esperar neste ponto antes de prosseguirem.

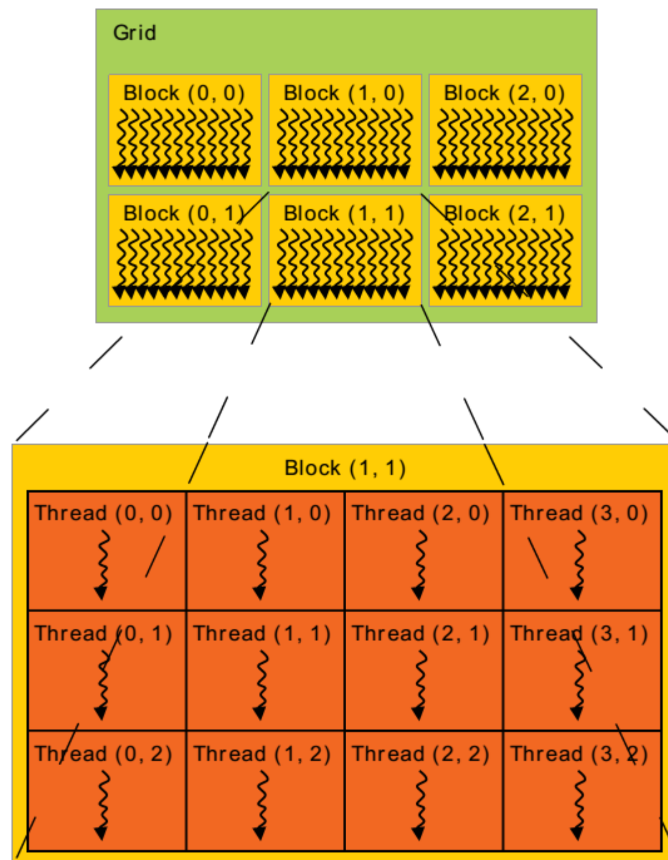

Figura 2.2: *Hierarquia de threads [26].*

2.3 Hierarquia de Memória

Em CUDA existem vários espaços de memória disponíveis para as *threads* acessarem durante sua execução, como podemos ver na Figura 2.3. Cada *thread* possui

uma memória local, que reside na memória global do dispositivo, e tem acesso privado à ela. Cada bloco conta com uma memória compartilhada que se encontra no chip, e é visível para todas *threads* que a ele pertencem. Durante a execução de um *kernel*, todas as *threads* tem acesso à uma mesma memória global que engloba a maior fatia da memória do dispositivo.

Existem ainda dois outros espaços de memória, que são somente de leitura, acessíveis por todas *threads*: a memória constante e de textura. Essas memórias e a global não são apagadas entre chamadas de *kernel* pela mesma aplicação. Como essa memória é acessível apenas por funções de GPU, a chamamos de memória do tipo *device* (dispositivo). As memórias alocadas na RAM que são visíveis para a CPU chamamos de *host* (hospedeiro). Caso seja necessária a transferência de dados entre *host* e *device*, o CUDA API oferece funções que facilitam esse processo, como alocação dinâmica de memória e cópia. Em relação à velocidade de acesso dessas memórias, a global fica fora do *chip* do multiprocessador e logo possui alta latência, enquanto a compartilhada que fica no *chip* possui baixa latência, semelhante aos registradores. Variáveis compartilhadas podem ser declaradas com o especificador `__shared__` dentro do *kernel*.

A memória global é acessada por transações de memória de 32, 64 ou 128 *bytes*. Estas transações devem ser alinhadas naturalmente, de forma que cada segmento de 32, 64 ou 128 estejam com seu primeiro endereço sendo múltiplo de seus tamanhos. Somente estes segmentos podem ser lidos ou escritos por transações de memória.

Quando a memória global é acessada por uma *warp*, os acessos das *threads* dentro da *warp* são aglutinados (*coalesced*) em uma ou mais dessas transações de memória, dependendo do tamanho da palavra acessada por cada *thread* e da distribuição de endereços de memória em todas *threads*. Assim, é importante escrever o *kernel* de maneira que possamos ter o proveito dessas transações de memória. Por exemplo, se uma transação de memória de 32 *bytes* for gerada para cada leitura de 4 *bytes* de uma *thread*, o rendimento será dividido por 8, pois 28 *bytes* foram lidos sem utilidade. O comportamento dessas transações de memória variam de acordo com a capacidade computacional do dispositivo. Veremos isso na Seção 2.4.

A organização dos dados é um aspecto que pode ajudar a garantir o máximo proveito do acesso aglutinado. Um exemplo simples é passar um vetor de estruturas para estrutura de vetores, fazendo com que todas *threads* acessem uma região de memória contígua.Outro exemplo, é fazer arredondamento de vetores bidimensionais para que tenham seu tamanho de largura e altura múltiplos do tamanho do *warp*.

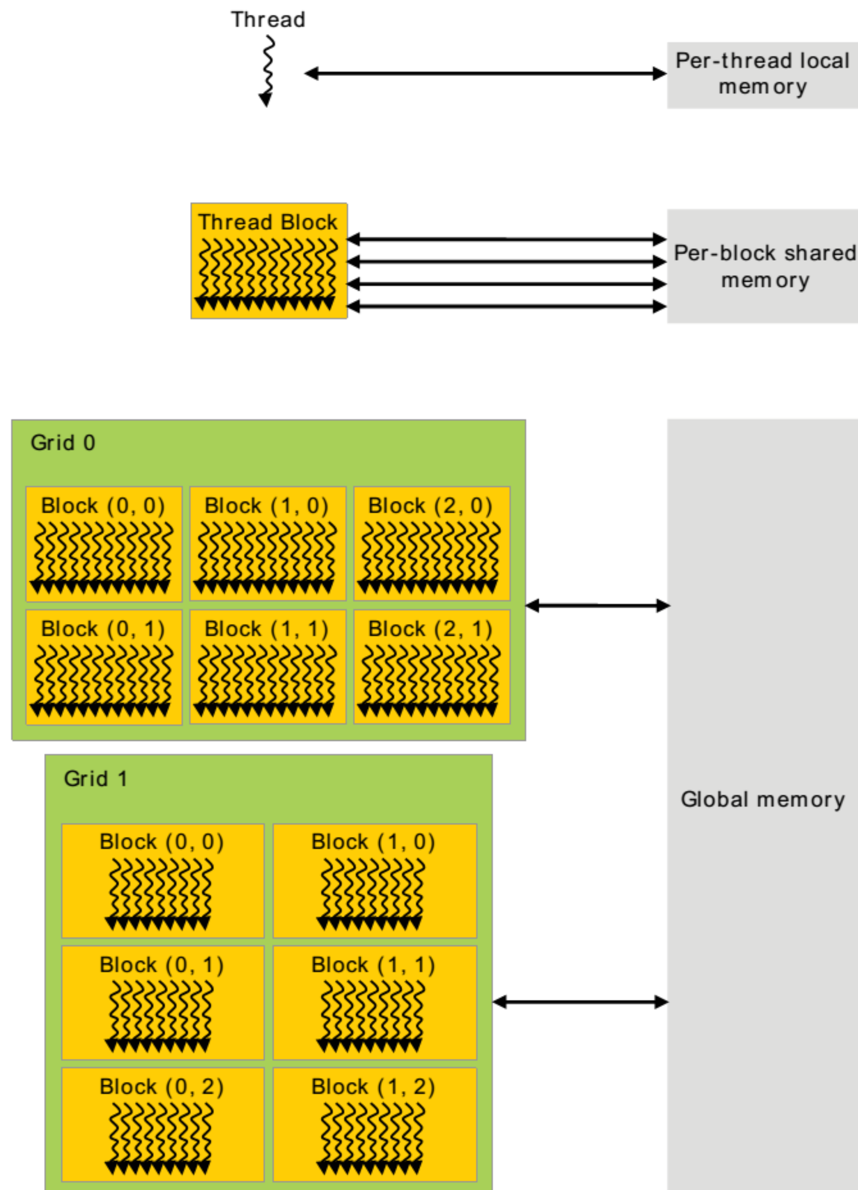

Figura 2.3: *Hierarquia da memória [26].*

2.4 Arquitetura Kepler

Neste trabalho, o CUDA Toolkit versão 7.5[1] e a arquitetura Kepler de placas da NVIDIA foram utilizados. A capacidade computacional (*compute capability*) de um dispositivo CUDA é usada para indicar quais recursos e limitações o *hardware* da GPU possui. Pela Tabela 2.1[2] podemos ver os recursos disponíveis da capacidade 3.5[3] da

[1]Disponível em https://developer.nvidia.com/cuda-75-downloads-archive.

[2]http://developer.download.nvidia.com/compute/cuda/CUDA_Occupancy_calculator.xls

[3]Durante a elaboração deste trabalho, a versão 9.0 do Toolkit está disponível, e dará suporte ao *compute capability* 7.0 das placas da arquitetura Volta. O guia mais recente encontra-se em: http://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html.

arquitetura Kepler. Há outros recursos e limitações como a disponibilidade de certas funções da API, que podem ser consultadas nos guias da NVIDIA.

Tabela 2.1: *Recursos computacionais da capacidade computacional 3.5.*

Recurso	Disponibilidade
Threads por Warp	32
Warps por Multiprocessador	64
Blocos de thread por Multiprocessador	16
Threads por Multiprocessador	2048
Threads por Bloco	1024
Registradores por Multiprocessador	65536
Registradores por Thread	255
Memória Compartilhada por Multiprocessador (bytes)	49152
Memória Compartilhada por Bloco	49152
Tamanho do Grupo da Alocação de Registradores	256
Granularidade da Alocação de Registradores	warp

Dispositivos de capacidade computacional 3.5 possuem 192 CUDA *cores* para cada SM e 4 escalonadores de *warp*. As *warps* são distribuídas entre os escalonadores, e cada escalonador emite 2 instruções independentes por vez para uma de suas *warps* assinaladas. A Figura 2.4 ilustra um escalonador da arquitetura Kepler. Estes dispositivos também possuem uma *cache* para a memória constante, além de uma *cache* L1 para cada SM e *cache* L2 compartilhada por todos SMs. A L1 é usada para fazer cache de acessos à memória local, incluindo registradores "transbordados"(*Spilled*) temporariamente, isto é, quando o número de registradores por *thread* passa do limite (255 nesse caso, como visto na Tabela 2.1) e são armazenados nessa *cache*. A L2 faz *cache* de acessos à memória local e global. Em alguns dispositivos é possível forçar a L1 a fazer *cache* da memória global também. A *cache* L1 e a memória compartilhada ficam no mesmo espaço no *chip* do SM. Pode-se mudar a configuração para 48KB para uma e 16KB para a outra, ou 32KB para ambas antes de lançar o *kernel*. A *cache* L2 conta com 1,5MB. Há ainda 48KB de uma *cache* para dados com apenas permissão de leitura (*read-only*), que pode ser acessada diretamente ou por meio das funções de acesso da memória de textura. Veja o fluxo de acesso na Figura 2.5.

A memória compartilhada conta com 32 bancos organizados de maneira que palavras de 32 *bits* (no modo 32 *bits*) sucessivas são mapeadas em banco sucessivos. Quando múltiplas *threads* de uma *warp* acessam um endereço de uma sub-palavra qualquer dentro da mesma palavra de 32 *bits*, a palavra é *broadcasted* para todas *threads* requerentes. Também é *broadcasted* se for dentro de duas palavras 32 bits, desde que seus índices sejam alinhados em 64 bits. Na escrita, apenas uma das *threads* escreve cada sub-palavra (a *thread* escolhida é indefinida). Quando acessam palavras de 32 *bits* diferentes

no mesmo banco, o acesso é serializado, e dá-se o nome de conflito de banco. No modo 64 bits, apenas não é possível fazer o acesso em duas palavras diferentes, e as outras definições se mantém.

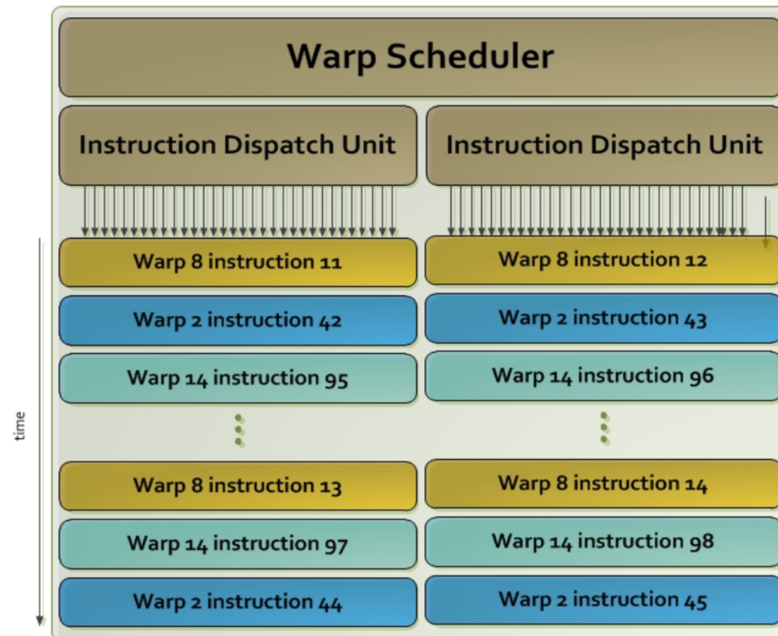

Figura 2.4: *Escalonador de warp na arquitetura Kepler [24].*

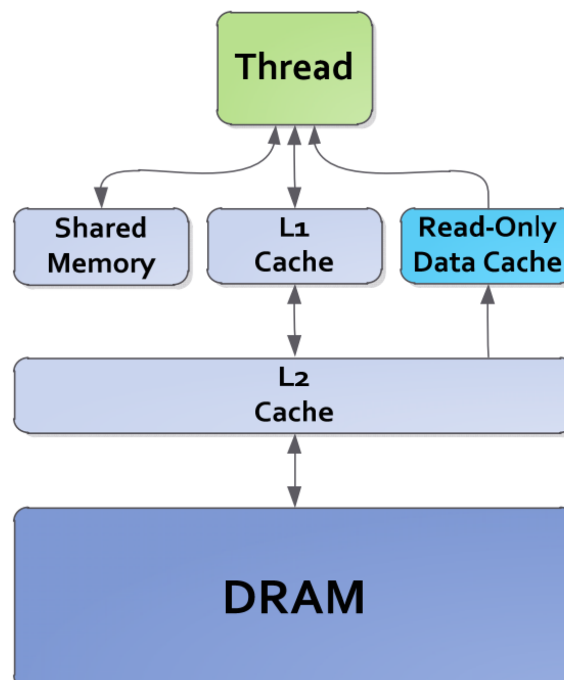

Figura 2.5: *Hierarquia de memória da arquitetura Kepler [24].*

Continuando sobre a memória global na Seção 2.3, os acessos à memória global são *cached* em linhas de *cache* de 128 *bytes* e são mapeados em segmentos alinhados de 128 *bytes* na memória do dispositivo. Acessos de memória que estão em *cache* em ambas L1 e L2, são servidos com uma transação de 128 *bytes* (Quando a opção de forçar o uso da L1 é utilizada), e quando somente na L2, de 32 *bytes* (Padrão na Kepler). Na Figura 2.6 podemos ver um exemplo de acesso aglutinado. Na primeira tabela, 32 acessos de memória de uma *warp* são transformados em apenas uma transação de 128 *bytes* se estiver em *cache*, ou em 4 de 32 bytes caso contrário. Estes acessos não precisam ser sequenciais, porém o máximo proveito desse recurso necessita do alinhamento de endereços, como visto na terceira tabela, uma transação de memória a mais é feita somente para a última *thread* que passou do alinhamento.

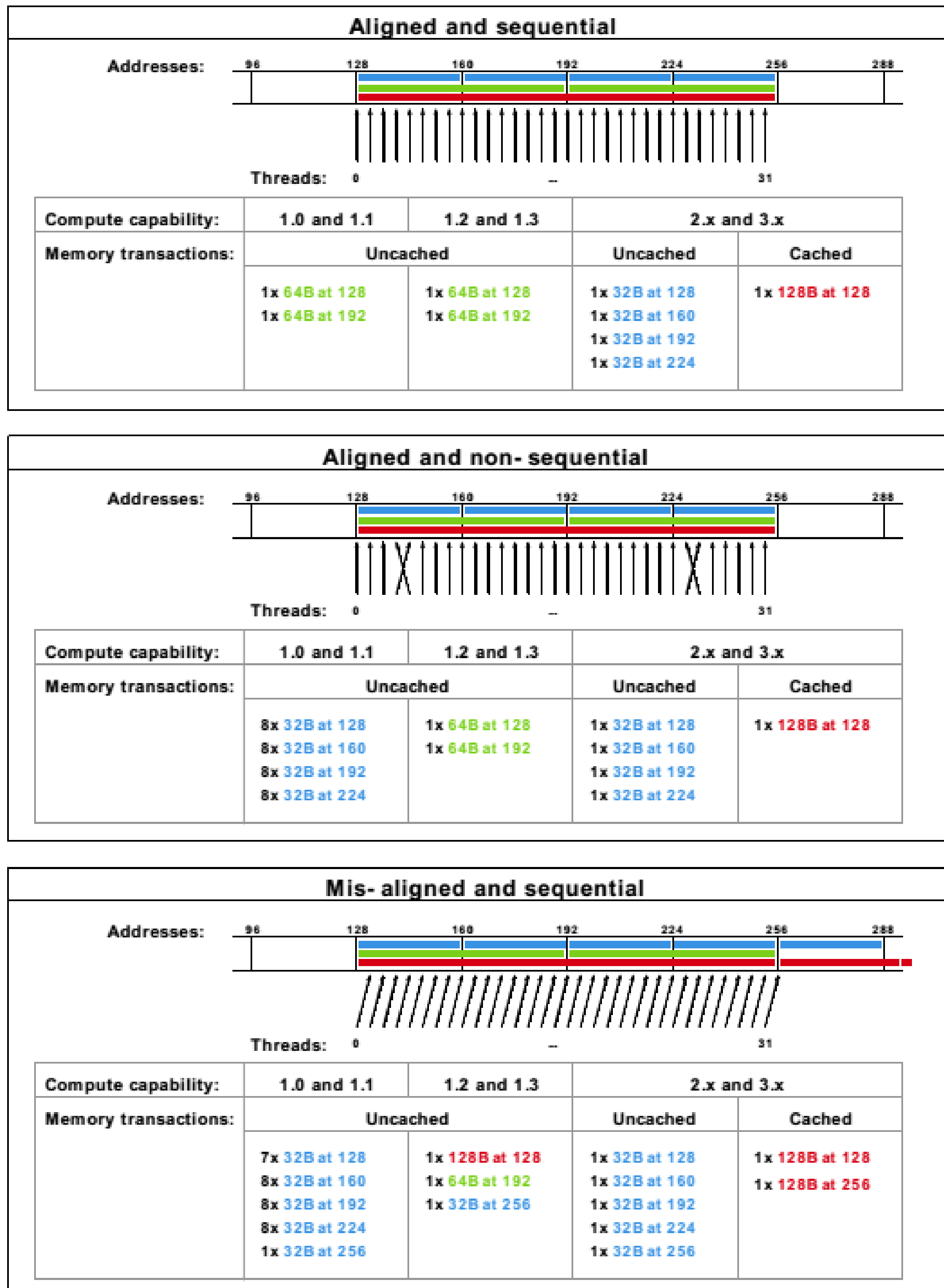

Figura 2.6: *Exemplos de acesso na memória global [26].*

Alguns dispositivos de capacidade computacional 2.0 ou acima podem executar múltiplos *kernels* concorrentemente. O limite é de até 32 *kernels* concorrentes na Kepler, e é feito com o uso de CUDA *streams* (fluxos). Com esse recurso podemos manter a utilização da GPU sempre alta e reduzirmos o tempo de execução total como mostra a

Figura 2.7 de Mark Harris. Outra vantagem é que permite desenvolvermos soluções com mais abordagens diferentes, como separar um problema em subproblemas e processá-los em *kernels* otimizados para cada um, desde que sejam independentes.

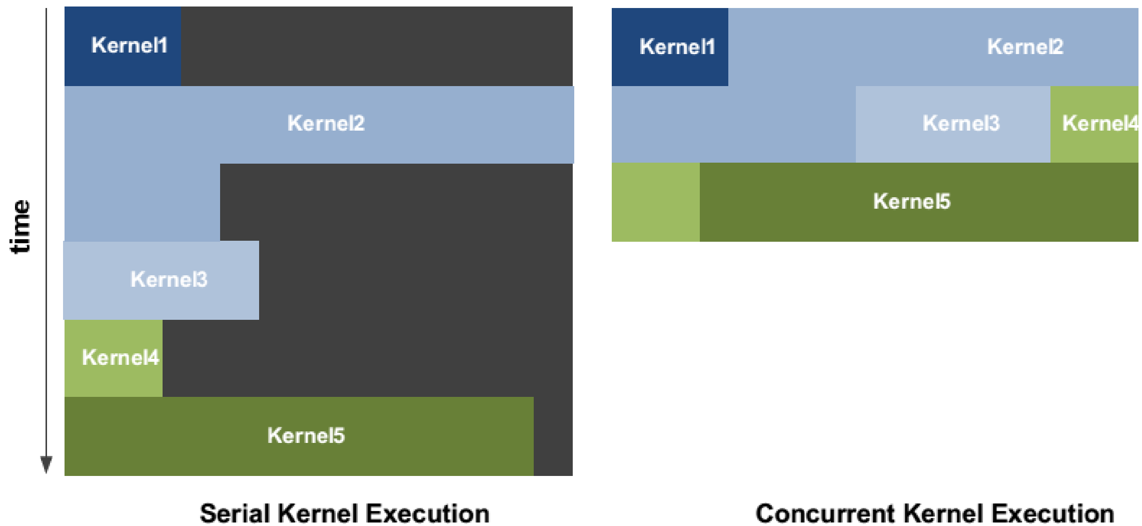

Figura 2.7: *Execução sequencial de kernel e execução com concorrência [26].*

Os *kernels* concorrentes também possibilitam a sobreposição de transferência de dados e processamento, com o uso de *streams* e do lançamento assíncrono dos *kernels*. A Figura 2.8 ilustra a comparação entre uma versão onde transferência, processamento, e recebimento de dados são feitos sequencialmente, e outra com 4 *streams*, mostrando que a sobreposição dessas tarefas diminui o tempo de solução.

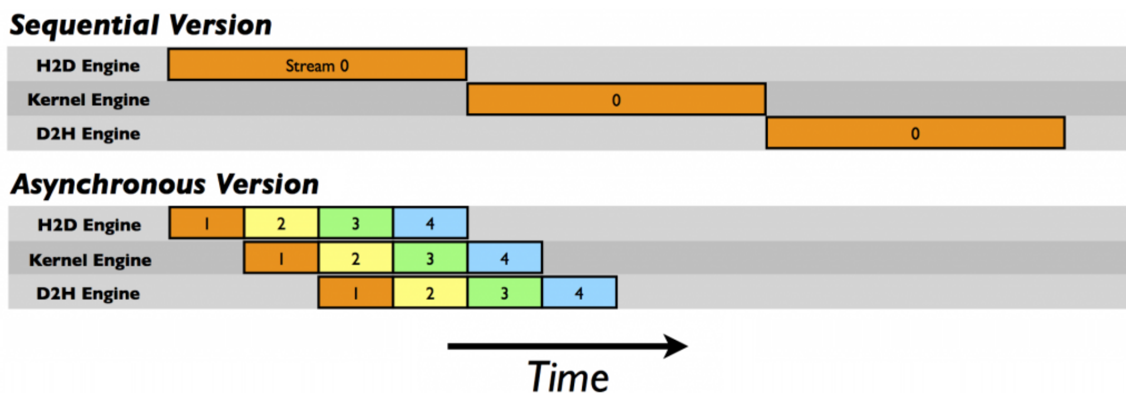

Figura 2.8: *Sobreposição de transferência (Fonte: Mark Harris)[15].*

2.5 Redução em CUDA

A redução é uma função aplicada sobre uma estrutura de dados recursiva, utilizando um único operador (soma, por exemplo) entre os dados de maneira que tenhamos

no final, um único valor resultado que representa alguma característica de todos elementos que a estrutura de dados continha. A natureza dessa função permite que apliquemos a função em paralelo em várias partições da estrutura de dados, reduzindo as partições em cada iteração. Em CUDA podemos copiar e/ou armazenar os dados na memória compartilhada de rápido acesso e aplicar a redução utilizando várias *threads*. Como um *warp* contém 32 *threads*, podemos criar um *kernel* que aproveita essa característica.

Vejamos o código 2.2 de Mark Harris [16] como exemplo de uma redução por soma. O *kernel* recebe como entrada o tamanho do bloco no *template*, um vetor de entrada `g_idata`, um vetor de saída `g_odata` e o tamanho do vetor. Este *kernel* é otimizado para tamanhos de bloco múltiplos de 32, e o tamanho do vetor múltiplo do tamanho do bloco. Na linha 5 calcula-se o índice relativo para as *threads* lerem posições diferentes. Uma soma inicial é feita na linha 8, checando os limites do vetor, e na linha 9 uma sincronização para todas *warps*. Da mesma maneira, as linhas 11-19 fazem cada vez menos *warps* reduzirem os dados e sincronizar. Nas linhas 20-26 não precisamos mais da barreira, já que uma *warp* contém 32 *threads* que executam o mesmo código. E para melhorar a performance, estas linhas usaram a técnica de *loop unrolling* (desenrolar de laços). O resultado final estará na primeira posição da memória compartilhada, e pode ser copiada por apenas uma *thread* para a variável de saída.

Podemos ver pela Figura 2.9 de Mark Harris como a redução comporta e possui complexidade logarítmica $O(\log_2 N)$. Considerando 16 elementos e 8 *threads*, no primeiro passo cada *thread* soma as partes do vetor que tem seu índice, com as partes de 8 somado a esse mesmo índice. No segundo passo, apenas metade das *threads* continuam a soma, agora saltando 4 índices. As próximas iterações vão reduzindo o trabalho pela metade, até que no final há somente o resultado final. Tendo assim, um número logarítmico de passos.

Código 2.2 `void reduce6()` [16]

```
template <unsigned int blockSize>
__global__ void reduce6(int *g_idata, int *g_odata, unsigned int n){
  extern __shared__ int sdata[];
  unsigned int tid = threadIdx.x;
  unsigned int i = blockIdx.x*(blockSize*2) + tid;
  sdata[tid] = 0;

  if (i < n) { sdata[tid] += g_idata[i] + g_idata[i+blockSize]; }
  __syncthreads();

  if (blockSize >= 512) {
    if (tid < 256) { sdata[tid] += sdata[tid + 256];
    } __syncthreads(); }
  if (blockSize >= 256) {
   if (tid < 128) { sdata[tid] += sdata[tid + 128];
    } __syncthreads(); }
  if (blockSize >= 128) {
   if (tid <  64) { sdata[tid] += sdata[tid +  64];
    } __syncthreads(); }
  if (tid < 32) {
    if (blockSize >= 64) sdata[tid] += sdata[tid + 32];
    if (blockSize >= 32) sdata[tid] += sdata[tid + 16];
    if (blockSize >= 16) sdata[tid] += sdata[tid + 8];
    if (blockSize >=  8) sdata[tid] += sdata[tid + 4];
    if (blockSize >=  4) sdata[tid] += sdata[tid + 2];
    if (blockSize >=  2) sdata[tid] += sdata[tid + 1];
  }
  if (tid == 0) g_odata[blockIdx.x] = sdata[0];
}
```

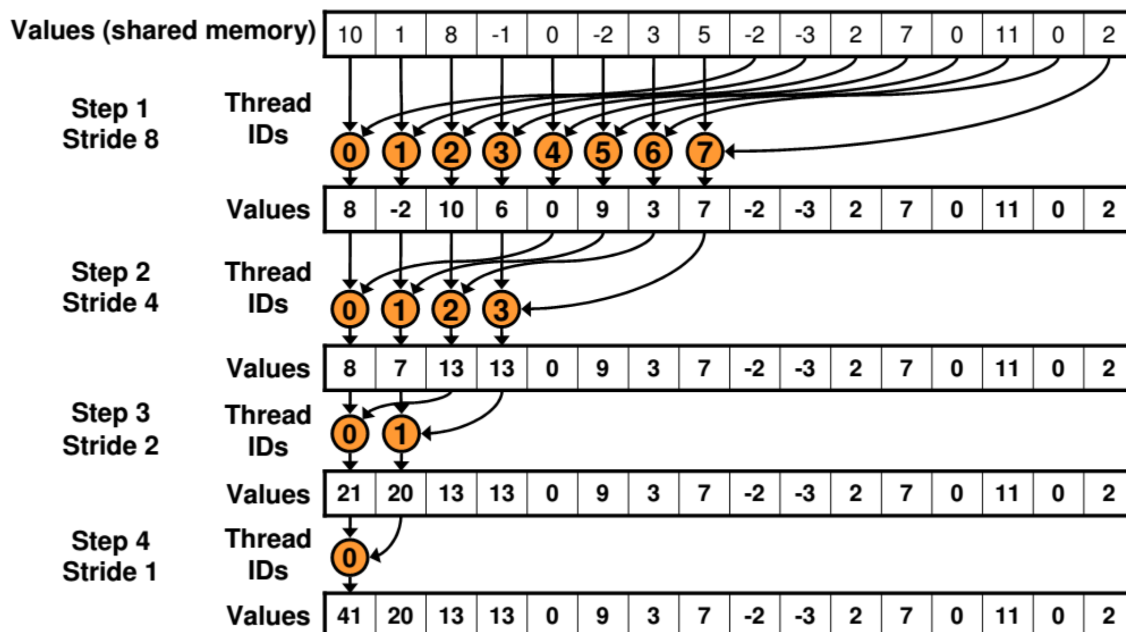

Figura 2.9: *Exemplo de redução por soma de um vetor usando CUDA [25].*

CAPÍTULO 3

Fundamentação Teórica e Trabalhos Relacionados

Neste capítulo detalhamos a fundamentação teórica de cada uma das três técnicas e fazemos um levantamento de trabalhos relacionados para cada um utilizado nesse trabalho. Na seção 3.1 detalhamos sobre *itemsets* frequentes e regras de associação, que foram usadas nos trabalhos base das técnicas de *learning to rank* e aprendizado ativo. Na Seção 3.3 apresentamos a técnica *learning to rank* e na Seção 3.2 discorremos sobre aprendizado ativo. Na Seção 3.4 apresentamos sobre a solução de k-vizinhos mais próximos (k-NN) para textos.

3.1 Itemsets Frequentes e Regras de Associação

Nesta seção encontra-se a fundamentação teórica para a mineração de *itemsets* frequentes (FIM - *Frequent Itemset Mining*) e regras de associação. FIM é utilizado para descobrir a co-ocorrência entre 2 ou mais objetos de acordo com sua frequência. O exemplo típico de aplicação é a análise de carrinho de compras, onde as compras de clientes de uma loja são minerados para descobrir quais produtos são comprados com frequência [42]. Pode-se ainda utilizar os itens frequentes minerados para calcular a ocorrência condicional de 2 conjuntos de itens distintos, usando o conceito de regras de associação. Essas regras nos fornecem informações do tipo "Se um cliente comprar carne, ele provavelmente irá comprar cerveja junto".

O problema de mineração de regras de associação foi introduzido por Agrawal et al. em [2]. Seguiremos as definições, tópicos e exemplos de Zaki e Meira Jr [42] como referência.

3.1.1 Representação do Banco de Dados e Itemsets

Temos $I = \{x_1, x_2, \ldots, x_m\}$ um conjunto de itens, onde um conjunto $X \subseteq I$ é chamado de *itemset*. Um *itemset* de cardinalidade k é chamado de *k-itemset*. Temos $\tau =$

$\{t_1, t_2, ..., t_n\}$, outro conjunto chamado de identificadores de transação ou $tids$ ($transaction$ $identifiers$). Um conjunto $T \subseteq \tau$ é chamado de $tidset$. É importante notar que é assumido que os $itemsets$ e $tidsets$ são mantidos em ordem lexicográfica. Uma transação é uma tupla na forma $\langle t, X \rangle$, onde $t \in \tau$ é um identificador único de transação.

Define-se um banco de dados binário D como uma relação binária entre os conjuntos de $tids$ e items, isto é, $D \subseteq T \times I$. Temos que $(t, x) \in D$ sse $x \in X$ na tupla $\langle t, X \rangle$. É dito que uma transação identificado por tid t contém o $itemset$ $X = \{x_1, x_2, ..., x_k\}$ sse $(t, x_i) \in D$ para todo $i = 1, 2, ..., k$.

A Figura 3.1(a) ilustra um banco de dados binário como exemplo. Aqui, $I = \{A, B, C, D, E\}$, e $T = \{1, 2, 3, 4, 5\}$. No banco binário, a posição na linha t, coluna x é 1 sse $(t, x) \in D$, e 0 caso contrário. Podemos observar que a transação 1 contém o item B e também contém o $itemset$ BE, e assim por diante.

Define-se uma função $\mathbf{i}$ que mapeia $tidsets$ para $itemsets$, como:

$$\mathbf{i}(T) = \{x \mid \forall t \in T,\ t \text{ contém } x\}$$

onde $T \subseteq \tau$, e $\mathbf{i}(T)$ é o conjunto de itens que são comuns para todas transações no $tidset$ T. Dessa forma pode-se considerar o banco de dados binário D como um banco de dados de transações consistindo de tuplas na forma $\langle t, \mathbf{i}(t) \rangle$. O banco de dados de transações (Figura 3.1(b)) é considerado como uma representação horizontal do banco de dados binário.

Define-se outra função $\mathbf{t}$ que mapeia $itemsets$ para $tidsets$, como:

$$\mathbf{t}(X) = \{t \mid t \in \tau \text{ e } t \text{ contém } X\}$$

onde $X \subseteq I$, e $\mathbf{t}(X)$ é o conjunto de $tids$ que contém todos os itens do $itemset$ X. Dessa forma pode-se considerar o banco de dados binário D como um banco de dados de $tidsets$ consistindo de tuplas na forma $\langle x, \mathbf{t}(x) \rangle$, onde $x \in I$. O banco de dados de $tidset$ é uma representação vertical (Figura 3.1(c)), e o binário também é uma forma vertical alternativa. A representação vertical permite que a composição de itens possa ser feita com operações de interseção, tendo apenas os $tids$ comuns no resultado.

A Figura 3.1(b) mostra o banco de dados de transação correspondente para o banco binário na Figura 3.1(a). Por exemplo, a primeira transação é $\{1, \{A, B, D, E\}\}$, onde omitimos o item C, já que $(1, C) \notin D$. Por conveniência, vamos escrever sem a notação de conjunto, como $\{1, ABDE\}$. A Figura 3.1(c) ilustra o banco vertical correspondente para o banco binário da Figura 3.1(a). Por exemplo, a tupla correspondendo ao item A, mostrado na primeira coluna, é $\{A, 1345\}$; omitimos $tids$ 2 e 6 porque $(2, A) \notin D$ e $(6, A) \notin D$.

D	A	B	C	D	E
1	1	1	0	1	1
2	0	1	1	0	1
3	1	1	0	1	1
4	1	1	1	0	1
5	1	1	1	1	1
6	0	1	1	1	0

(a) Binary database

t	$\mathbf{i}(t)$
1	$ABDE$
2	BCE
3	$ABDE$
4	$ABCE$
5	$ABCDE$
6	BCD

(b) Transaction database

x	A	B	C	D	E
$\mathbf{t}(x)$	1	1	2	1	1
	3	2	4	3	2
	4	3	5	5	3
	5	4	6	6	4
		5			5
		6			

(c) Vertical database

Figura 3.1: *a) Banco de dados binário. b) Banco de dados horizontal. c) Banco de dados vertical [42].*

O suporte de um *itemset* é o número de transações em que ele ocorre. Define-se o suporte de um *itemset* X em um banco de dados D como $sup(X,D)$:

$$sup(X,D) = |\{t \mid \langle t, \mathbf{i}(t)\rangle \in D \text{ e } X \subseteq \mathbf{i}(t)\}| = |\mathbf{t}(X)|$$

Define-se o suporte relativo de X, que é a fração de transações em que X ocorre: $rsup(X,D) = \frac{sup(X,D)}{|D|}$

Assim, dado um limiar de suporte mínimo *minsup*, o *itemset* X será frequente em D se $sup(X,D) \geq minsup$ e se *minsup* for um inteiro, ou $rsup(X,D) \geq minsup$ se for uma fração. Passaremos a escrever apenas $sup(X)$ e $rsup(X)$. Dessa maneira temos $\mathcal{F}$, o conjuntos de todos *itemsets* frequentes e $\mathcal{F}^{(k)}$ para os *k-itemsets*.

Dado o banco de dados de exemplo na Figura 3.1, considere *minsup* = 3 (ou *minsup* = 0,5, se usarmos $rsup(X)$ para esse exemplo). A tabela na Figura 3.2 mostra todos os 19 *itemsets* frequentes no banco de dados, agrupados por seu valor de suporte. Por exemplo, o *itemset* BCE está contido nos *tids* 2, 4 e 5, assim $\mathbf{t}(BCE)$ = 245 e $sup(BCE)$ = $|\mathbf{t}(BCE)|$ = 3. O que quer dizer que BCE é um *itemset* frequente. Os 19 *itemsets* frequentes mostrados na tabela formam o conjunto $\mathcal{F}$. Os conjuntos de todos os *k-itemsets* frequentes são

$$\mathcal{F}^{(1)} = \{A, B, C, D, E\}$$
$$\mathcal{F}^{(2)} = \{AB, AD, AE, BC, BD, BE, CE, DE\}$$
$$\mathcal{F}^{(3)} = \{ABD, ABE, ADE, BCE, BDE\}$$
$$\mathcal{F}^{(4)} = \{ABDE\}$$

3.1.2 Mineração de Itemset e de Regras de Associação

Uma regra de associação é uma expressão do tipo $X \longrightarrow Y$, onde X é chamado de antecedente e Y de consequente. Ela representa uma relação de co-ocorrência entre X e Y, ou mais especificamente, como Y co-ocorre quando temos X. Por definição, X e Y

Table 8.1. Frequent itemsets with $minsup = 3$

sup	itemsets
6	B
5	E, BE
4	$A, C, D, AB, AE, BC, BD, ABE$
3	$AD, CE, DE, ABD, ADE, BCE, BDE, ABDE$

Figura 3.2: *Itemsets frequentes da Figura 3.1 [42].*

devem ser disjuntos e o conceito de suporte também se aplica. Uma regra de associação é definida como uma expressão $X \longrightarrow Y$, onde X e Y são *itemsets* disjuntos, isto é $X, Y \subseteq I$, e $X \cap Y = \emptyset$. O *itemset* representado por $X \cup Y$ é denotado por XY. O suporte da regra é o número de transações em que XY ocorre:

$$s = sup(X \longrightarrow Y) = |\mathbf{t}(XY)| = sup(XY)$$

Do mesmo modo, temos o suporte relativo da regra, e serve como uma estimativa da probabilidade conjunta de X e Y:

$$rsup(X \longrightarrow Y) = \frac{sup(XY)}{|D|} = P(X \wedge Y)$$

Temos o conceito de confiança de uma regra, que é a probabilidade condicional de uma transação conter Y dado que contém X:

$$c = conf(X \longrightarrow Y) = P(Y|X) = \frac{P(X \wedge Y)}{P(X)} = \frac{sup(XY)}{sup(X)}$$

Dessa forma, uma regra é frequente se $sup(XY) \geq minsup$, e uma regra é chamada de forte se $conf \geq minconf$, onde $minconf$ é um limiar de confiança mínima.

Considere a regra de associação $BC \longrightarrow E$. Utilizando os valores de suporte dos *itemsets* mostrados na Figura 3.2, o suporte e confiança da regra são demonstrados abaixo:

$$s = sup(BC \longrightarrow E) = sup(BCE) = 3$$
$$c = conf(BC \longrightarrow E) = \frac{sup(BCE)}{sup(BC)} = \frac{3}{4} = 0,75$$

Como o conceito de suporte se aplica às regras de associação, é necessário minerar os *itemsets* frequentes em primeiro lugar de acordo com um valor de *minsup*. Estes *itemsets* frequentes irão compor o conjunto $\mathcal{F}$, e a partir deste conjunto, tendo um valor de confiança mínimo *minconf*, podemos minerar as regras frequentes e fortes. Assim, mostraremos primeiramente alguns algoritmos de mineração de *itemsets* frequentes.

A quantidade de *itemsets* possíveis usando os itens de I é $2^{|I|}$. Para mostrar o tamanho desse problema, começaremos descrevendo um algoritmo força-bruta que enumera todos *itemsets* $X \subseteq I$. Para cada *itemset*, seu respectivo suporte é calculado de acordo com o banco de dados de entrada D. O algoritmo é dividido em dois passos: geração de candidatos e cálculo de suporte.

No primeiro passo são gerados todos $2^{|I|}$ *itemsets* possíveis, e cada um é chamado de candidato, pois são candidatos a serem *itemsets* frequentes. Para ilustrar o espaço de busca exponencial desses *itemsets*, pode-se criar uma grade como visto na Figura 3.3.

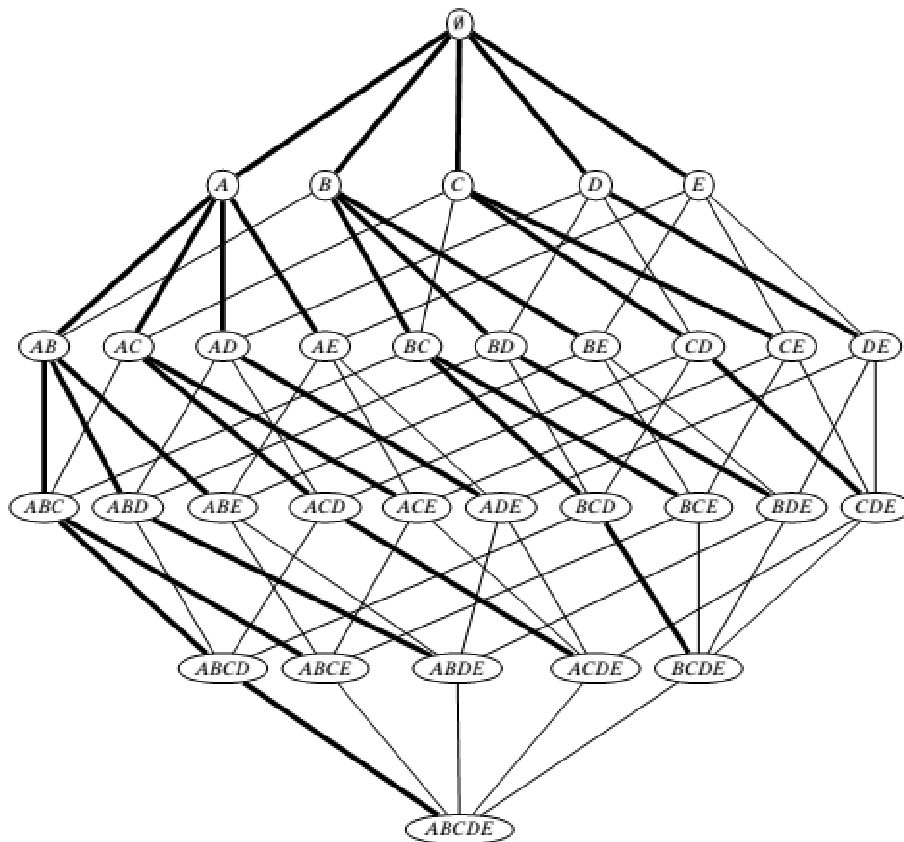

Figura 3.3: *Grade de itemsets, onde o traço em negrito representa a árvore de prefixo [42].*

A Figura 3.3 ilustra a grade dos *itemsets* para o conjunto de itens $I = \{A, B, C, D, E\}$. Existem $2^{|I|} = 2^5 = 32$ possíveis *itemsets*, incluindo o vazio. O método por força-bruta explora todo o espaço de busca do *itemset*, independente do limite do *minsup* empregado. Se *minsup* = 3, então o método de força-bruta teria como resultado o conjunto de *itemsets* frequentes mostrados na Figura 3.2.

No segundo passo é determinado se o *itemset* candidato X é frequente. O algoritmo por força-bruta encontra-se no Algoritmo 1. Na linha 1 o conjunto $\mathcal{F}$ de *itemsets* frequentes é inicializado. Na linha 2 é enumerado cada *itemset* X possível de I. Na linha 3 o suporte para o *itemset* X é calculado usando o banco de dados D. Nas linhas

4-5 o limiar do suporte é checado com $minsup$, e o $itemset$ é inserido em $\mathcal{F}$ se atender o limiar. Na linha 6 o conjunto $\mathcal{F}$ é retornado. Nas linha 7-10, para cada transação no banco de dados D, é checado se X é um subconjunto dessa transação. Se sim, seu suporte é incrementado em 1.

Algorithm 1 BruteForce - Algoritmo força bruta de mineração de $itemsets$ frequentes [42].

Require: Database D, Items set I, $minsup$
Ensure: The set of frequent itemsets $\mathcal{F}$

1: $\mathcal{F} = \emptyset$
2: **for all** $X \subseteq I$ **do**
3: $sup(X) = COMPUTESUPPORT(X,D)$
4: **if** $(sup(X) \geq minsup)$ **then**
5: $\mathcal{F} = \mathcal{F} \cup \{(X, sup(X))\}$
6: **return** $\mathcal{F}$

COMPUTESUPPORT(X,D):
7: $sup(X) = 0$
8: **for all** $\langle t, \mathbf{i}(t) \rangle \in D$ **do**
9: **if** $(sup(X) \subseteq \mathbf{i}(t))$ **then**
10: $sup(X) = sup(X) + 1$
11: **return** $sup(X)$

Percebe-se que o método força bruta desperdiça muito tempo em cálculos com $itemsets$ que não são frequentes. Mas para dois $itemsets$ X,Y quaisquer, se $X \subseteq Y$, então $sup(X) \geq sup(Y)$, pois $itemsets$ maiores (Y) ocorrem em menor ou mesma frequência do que $itemsets$ menores (X). Desse fato podemos tirar duas propriedades. Se X é frequente, então todos seus subconjuntos são frequentes também. E se X é infrequente, então todos seus superconjuntos são infrequentes também. Em [3], Agrawal et al. chama essas propriedades de propriedade *Apriori* e propriedade antimonotônica do suporte, respectivamente.

Usando essas duas propriedades, Agrawal et al. desenvolveram o Apriori [3], que segue uma exploração de busca em largura ou por níveis. O Apriori evita a geração dos superconjuntos de qualquer $itemset$ não frequente, ou de $itemsets$ que tenham subconjuntos não frequentes. Nessa abordagem por níveis, o *Apriori* cria os k-$itemsets$ para cada nível k, e calcula o suporte de todos esses k-$itemsets$ lendo o banco de dados uma única vez para cada nível k. Assim, a complexidade de E/S do *Apriori* é bem reduzida em relação ao método força bruta. O seu processamento também é representado como uma árvore de prefixos.

O *Apriori* pode ter o mesmo desempenho que o força bruta, pois há a possibilidade de todos $itemsets$ serem frequentes. Mas na prática, com a poda que é feita no espaço de busca, retirando os $itemsets$ infrequentes, o custo computacional é bem menor. Em termos de E/S (Entrada/Saída) são necessárias apenas k leituras do banco de dados, com k sendo o tamanho do maior $itemset$ frequente.

Para gerar regras de associação fortes, utiliza-se os *itemsets* frequentes $\mathcal{F}$ que foram adquiridos por algum algoritmo de mineração, o Apriori por exemplo. Dado um *itemset* $Z \in \mathcal{F}$, todos seus subconjuntos $X \subset Z$ são verificados para calcular regras da forma $X \xrightarrow{s,c} Y$, onde $Y = Z \backslash X$ ou $Z - X$, ou seja, $X \cap Y = \emptyset$. A regra já é frequente pois $s = sup(XY) = sup(Z) \geq minsup$. Só é necessário checar se sua confiança satisfaz $minconf : c = \frac{sup(X \cup Y)}{sup(X)} = \frac{sup(Z)}{sup(X)}$

Se $c \geq minconf$, então a regra é forte. Pela confiança pode-se fazer uma poda na mineração de regras. Se $conf(X \longrightarrow Y) < minconf$, então $conf(W \longrightarrow Z \backslash W) < minconf$ para todos subconjuntos $W \subset X$, já que $sup(W) \geq sup(X)$ por W ter menor cardinalidade. Pode-se então podar a checagem dos subconjuntos de X.

O Algoritmo 2 descreve o método da mineração de regras de associação. Na linha 1, retira-se um *itemset* frequente Z de $\mathcal{F}$ de cardinalidade maior do que 2, pois esse é o tamanho mínimo para criar uma regra. Na linha 2, é colocado em A, os antecedentes, todos subconjuntos de Z menores que $|Z|$, exceto o conjunto vazio. Na linha 3 inicia o processamento de cada item de A. Na linha 4-5, o maior elemento de A é escolhido e retirado. Na linha 6 a confiança é calculada e nas linhas 7-8 é checada se satisfaz $minconf$, imprimindo a regra $X \longrightarrow Z - X$ se sim. Se não satisfazer, na linha 10 é feito a poda de todos subconjuntos de X, pois suas confianças também não irão satisfazer $minconf$. Então é escolhido o maior item na linha 4, para maximizar o efeito da poda.

Algorithm 2 AssociationRules - Algoritmo de mineração de regras de associação [42].

Require: The set of frequent itemsets $\mathcal{F}$, $minconf$
Ensure: The strong association rules

1: **for all** $Z \in \mathcal{F}$, such that $|Z| \geq 2$ **do**
2: $A = \{X \mid X \subset Z, X \neq \emptyset\}$
3: **while** $A \neq \emptyset$ **do**
4: X = maximal element in A
5: $A = A \backslash X$ // remove X from A
6: $c = sup(Z)/sup(X)$
7: **if** $(c \geq minconf)$ **then**
8: print $X \longrightarrow (Z - X), sup(Z), c$
9: **else**
10: $A = A \backslash \{W \mid W \subset X\}$ // remove all subsets of X from A

Considere o *itemset* frequente $ABDE(3)$ da tabela na Figura 3.1, cujos suportes são mostrados entre parênteses. Considere que $minconf = 0,9$. Para gerar regras de associação fortes, o conjunto de antecedentes é inicializado como

$A = \{ABD(3), ABE(4), ADE(3), BDE(3), AB(3), AD(4), AE(4), BD(4), BE(5), DE(3), A(4), B(6), D(4), E(5)\}$

O primeiro subconjunto é $X = ABD$, e a confiança de $ABD \longrightarrow E$ é $\frac{3}{3}$ = 1,0. O próximo subconjunto é $X = ABE$, mas a regra correspondente $ABE \longrightarrow D$ não é forte pois $conf(ABE \longrightarrow D) = \frac{3}{4} = 0,75$. Pode-se então remover de A todos os subconjuntos de ABE. O conjunto atualizado de antecedentes é então:

$A = \{ADE(3), BDE(3), AD(4), BD(4), DE(3), D(4)\}$

Depois, escolhemos $X = ADE$, que produz uma regra forte, assim como $X = BDE$ e $X = AD$. No entanto, ao processar $X = BD$, encontra-se $conf(BD \longrightarrow AE) = \frac{3}{4} = 0,75$, assim todos os subconjuntos de BD podem ser removidos de A, para resultar em :

$A = \{DE(3)\}$

A última regra a ser tentada é $DE \longrightarrow AB$, que também é forte. O conjunto final de regras fortes resultantes são:

$ABD \longrightarrow E, conf = 1,0$
$ADE \longrightarrow B, conf = 1,0$
$BDE \longrightarrow A, conf = 1,0$
$AD \longrightarrow BE, conf = 1,0$
$DE \longrightarrow AB, conf = 1,0$

3.1.3 Outros Trabalhos Relacionados

Agrawal et al. [2] definem as propriedades de suporte e confiança na área de mineração de regras de associação. É criada uma nova métrica de suporte esperado para evitar cálculos para *itemsets* maiores, e usa duas técnicas de poda para reduzir o espaço de busca. O algoritmo conseguiu manter a acurácia próxima a 100% usando o suporte esperado. Em [36, 43, 11] é utilizado GPU para acelerar o processo de mineração de *itemsets* frequentes. Teodoro et al. e Zhang et al. utilizam árvores na CPU para criar e organizar os *itemsets* candidatos, que são enviados para a GPU fazer a contagem do suporte de cada candidato. Teodoro et al. optaram por transformar o banco de dados em uma lista invertida compactada, enquanto Zhang et al. usaram mapa de bits para permitir operações bit a bit para facilitar o cálculo de suporte.

Djenouri et al. propuseram uma solução que utiliza o algoritmo evolucionário de otimização por enxame abelhas, e foram pioneiros em minerar *datasets* grandes com o seu método. A cada geração são criadas abelhas, onde é escolhida aquela com o melhor *fitness*, baseado num cálculo de suporte e confiança. O *fitness* é calculado na GPU, e foi desenvolvido um método onde uma única regra é avaliada na GPU por vez, e outro onde múltiplas regras são avaliadas. O primeiro método teve problemas com a constante transferência de memória entre CPU e GPU, e a segunda solução foi a melhor.

3.2 Aprendizado Ativo

Em um contexto de aprendizado de máquina, onde bases de treino rotuladas são criadas com a ajuda de humanos a partir de grandes bases de dados não-rotuladas, o Aprendizado Ativo tem como objetivo reduzir o esforço de rotulagem dos humanos. Por um processo iterativo, os humanos são ativamente consultados para rotular amostras selecionadas por um algoritmo de aprendizado de máquina, até que uma medida de erro ou de representatividade atinja limiares definidos [14].

Esta redução ajuda a remover ruídos e simplifica a criação de modelos, mantendo apenas informações relevantes [33]. Neste trabalho, o aprendizado ativo foi aplicado no contexto de *learning to rank*, onde as bases de dados têm as mesmas definições descritas na Seção 3.3.

3.2.1 Aprendizado Ativo com Regras de Associação

Silva et al. [33] usaram aprendizado ativo simulando a presença do usuário, utilizando os rótulos originais disponíveis no conjunto de treino. Propuseram um algoritmo de amostragem seletiva SSAR (Selective Sampling using Association Rules) que gera um subconjunto de treino $\mathcal{U}$ bem pequeno e representativo do conjunto de treino original $\mathcal{D}$.

O aspecto mais importante do SSAR é selecionar os documentos que compartilham o menor número de $features$ entre si. Consequentemente, $\mathcal{U}$ tem apenas documentos com uma sobreposição mínima de $features$. Neste contexto, uma sobreposição significa a projeção de um registro d em registros $\mathcal{D}$, chamado de $\mathcal{D}_d$. Definimos a projeção como uma seleção de registros de $\mathcal{D}$ que tenham o mesmo conjunto de $features$ ou com interseção diferente de $\emptyset$. Essa sobreposição entre $features$ de registros pode ser parcial ou completa, e é ajustada de acordo com o tamanho das regras de associação.

Descrito no Algoritmo 3, o algoritmo SSAR escolhe o registro d que maximiza o tamanho da projeção em $\mathcal{D}$ na linha 1, isto é, d é o documento para qual $|\mathcal{D}_d|$ é o maior possível (ou o mais representativo). Em contraste, para os próximos documentos a menor sobreposição possível é considerada. Isto encontra-se nas linhas 6-8, onde as regras de associação (definidas na próxima Seção 3.3.1) de cada documento são geradas usando o conjunto $\mathcal{U}$ atual. Na linha 6, o registro que gera a menor quantidade de regras é escolhido, indicando que é o mais dissimilar entre os registros que ainda não foram escolhidos. Na linha 8, o documento escolhido é inserido em $\mathcal{U}$.

A Figura 3.4 ilustra o SSAR de modo geral. Assumindo na figura que o documento inicial já esteja em na base reduzida U, o processo iterativo consiste em atualizar os *bitmaps* com o documento inserido recentemente, e então utilizá-los para gerar e contar as regras de associação de cada documento dos dados de treino D. Então a amostra d que gerou menos regras de associação é escolhida, onde o seu rótulo é obtido da base de

Algorithm 3 SSAR - Selective Sampling using Association Rules.

Require: The training data $\mathcal{D}$
Ensure: The reduced training data $\mathcal{U}$
Let $\mathcal{D}_{d_i}$ be the set of documents from the projection of document d_i on a training set $\mathcal{D}$
1: $y \Longleftarrow d_i$ such that $\forall d_j : |\mathcal{D}_{d_i}| \geq |\mathcal{D}_{d_j}|$
2: append y to $\mathcal{U}$
3: **loop**
4: **for all** document $d_i \in \mathcal{D}$ **do**
5: $\mathcal{R}_{d_i} \Longleftarrow$ rules extracted from $\mathcal{U}_{d_i}$
6: $y \Longleftarrow d_i$ such that $\forall d_j : |\mathcal{R}_{d_i}| \leq |\mathcal{R}_{d_j}|$
7: **if** $y \in \mathcal{U}$ **then break**
8: **else** append y to $\mathcal{U}$

Algorithm 4 SSARP - Selective Sampling using Association Rules with Partitions.

Require: The training data $\mathcal{D}$, Q partitions, Ordered features by χ^2 distribution $\mathcal{F}$
Ensure: The reduced training data $\mathcal{U}$
1: $\mathcal{U} = \emptyset$
2: **for all** $q \mid 0 \leq q < Q$ **do**
3: $i = 0$, $\mathcal{V} = \emptyset$
4: **while** $j = q + i \times Q \mid j < |\mathcal{F}|$ **do**
5: $\mathcal{V} = \mathcal{V} \cup \mathcal{F}[j]$
6: $i = i + 1$
7: $\mathcal{D}^q \Longleftarrow \mathcal{D}$ with columns of indexes in $\mathcal{V}$
8: $\mathcal{U} = \mathcal{U} \cup SSAR(\mathcal{D}^q)$

treino [1]. Caso essa amostra já esteja na base reduzida, o processo atingiu a convergência, e retorna a base reduzida. Se não estiver, é feita outra iteração.

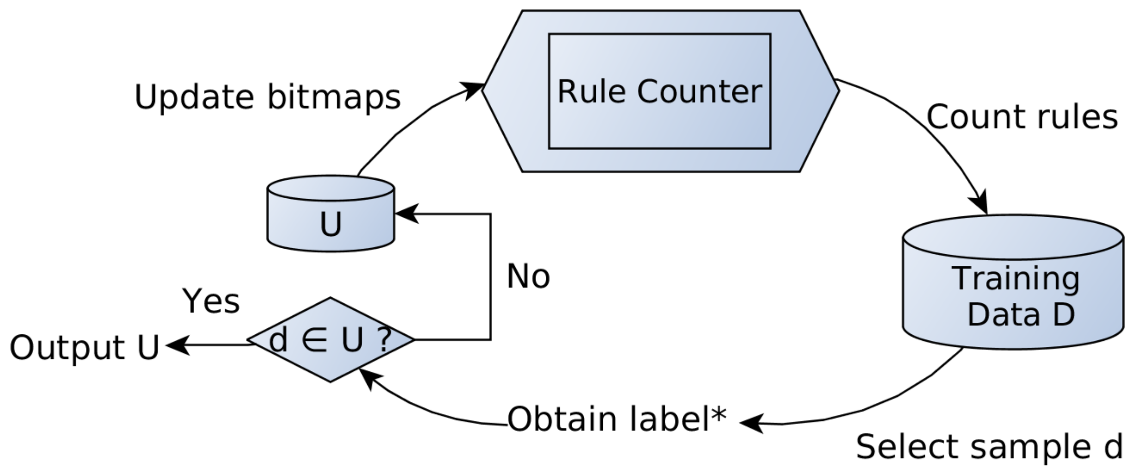

Figura 3.4: *Visão geral do SSAR.*

O conjunto reduzido produzido pelo SSAR é muito pequeno, mas tem pouca diversidade e isso afeta a performance do ranqueamento. Para resolver este problema, um esquema de particionamento vertical foi usado sobre as m *features* dos documentos de entrada, onde cada partição tem um subconjunto de cada documento com $\frac{m}{Q}$ *features*,

[1]Em um modo semi-supervisionado, com uma base não-rotulada, esse passo seria o de consultar o rótulo com um especialista.

considerando Q partições. Primeiro, as $features$ são ordenadas por sua distribuição χ^2, mostrando a relação de uma $feature$ para outra, com a intenção de agrupar as que são menos relacionadas para aumentar a diversidade. E então múltiplos conjuntos reduzidos são produzidos ao usar agrupamentos diferentes dessas $features$.

O Algoritmo 4 mostra como o particionamento é feito no SSAR *with Partitions* (SSARP). A mesma entrada é recebida como no SSAR, adicionado do número de partições e um vetor das $features$ ordenadas pelo χ^2. Nos testes de validação de Silva et al., o número de partições deve possuir entre 8 e 12 $features$ por partição. A linha 2 representa a iteração sobre cada partição. As partições são criadas nas linhas 4-6, de acordo com o vetor ordenado, onde as $features$ são distribuídas de maneira *round-robin* (linha 4) para todas partições. Nas linhas 7-8, o conjunto original é filtrado de acordo com as colunas baseadas nos índices das partições. Cada conjunto é mesclado no final, fazendo o conjunto final conter apenas os documentos únicos entre todas partições.

A Figura 3.5 ilustra o SSARP de modo geral, onde o SSAR é aplicado Q vezes. Cada partição tem um subconjunto disjunto de $features$ dos documentos, distribuídas de acordo com o valor χ^2, e cada partição converge em um número diferente de iterações com o SSAR. Como podem ter documentos iguais inseridos em suas bases reduzidas, é feita a operação de união entre o resultado de cada partição. E essa é o resultado do SSARP.

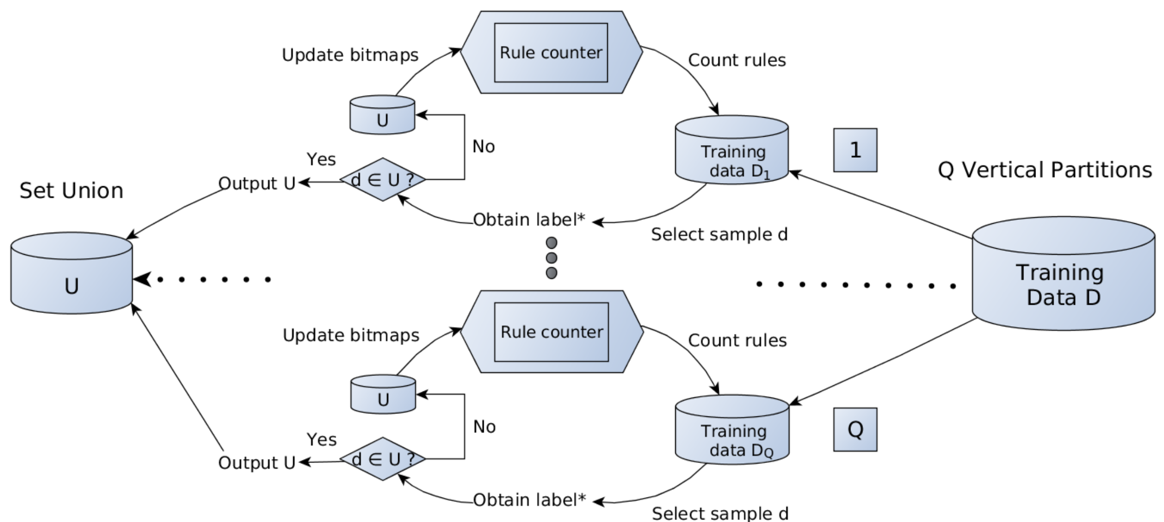

Figura 3.5: *Visão geral do SSARP.*

3.2.2 Outros Trabalhos Relacionados

Em [34], Silva et al. acrescenta ramum novo estágio no aprendizado ativo. O primeiro estágio é o SSARP, que ajuda o segundo estágio a convergir. O segundo estágio é uma estratégia de consulta por comitê (QBC - *Query Based Committee*), que seleciona

os 5 documentos em que os membros mais discordam de sua relevância, para cada consulta. Os membros do comitê são: SVMRank, RankBoost e LRAR. Os documentos selecionados são acrescentados na base reduzida, e essa é usada como treino para a próxima seleção. Nos testes foram feitas 25 rodadas de seleção, e com apenas 6% da base original, conseguiu um resultado competitivo aos com 100%.

Long et al. [20] criaram um método de aprendizado ativo que busca minimizar a métrica perda DCG (*Discounted Cumulative Gain*), aplicado para ranqueamento de buscas na Web. Num primeiro estágio é feito uma seleção de consultas mais informativas, e então, nessas são feitas seleções de documentos relevantes e até irrelevantes para dar diversidade. Seu método conseguiu uma redução de 20% a 64%, em relação à base original. Mehrotra e Yilmaz [23] propuseram um método de seleção de consultas baseado em informatividade e representatividade, onde as estratégias de probabilidade de permutação e modelo de tópico capturam esses dois aspectos respectivamente. Utilizaram funções submodulares para selecionar um subconjunto de consultas, baseada na incerteza e desacordo. Seu método consegue selecionar 5% das consultas e ainda ter um melhor NDCG@10 em alguns *datasets*.

3.3 Learning to Rank

Learning to Rank (L2R) trabalha com um conjunto de treino $\mathcal{D}$ composto por registros na forma $\langle q,d,r\rangle$, onde q é uma *query* (consulta), d é um documento e $r \in \{r_0,...,r_c\}$ é um valor de relevância de d para q. Um documento d é representado como uma lista de m *features* (características) $f_1, f_2, ..., f_m$. O conjunto $\mathcal{D}$ é usado para aprender uma função de ranqueamento ϕ, que mapeia as *features* de um documento para um valor de relevância r. Um conjunto de teste $\mathcal{T}$ é composto por registros na forma $\langle q,d,?\rangle$, onde a relevância r é desconhecida e será estimada pela função ϕ.

Para cada registro, há um documento representado por suas *features* e um valor de relevância associado com ele. As *features* dos documentos variam de simples fórmulas que usam a frequência de termos, para funções de ranqueamento proprietárias como o PageRank [27]. A função PageRank dá um número de ranqueamento para uma página web baseado no número de *links* que apontam para essa página. Para o *dataset* LETOR 3.0, algumas *features* usam $tf(q_t,d)$ e $idf(q_t)$ [28], onde $tf(q_t,d)$ representa o número de ocorrências do termo de *query* q_t no documento d, e $idf(q_t)$ é o *inverse document frequency* (IDF) (inverso da frequência nos documentos) daquele termo. O IDF dá um peso maior para os termos que raramente aparecem, e menor peso para os termos frequentes. A definição completa das *features* encontram-se na especificação da LETOR [28].

Damos um exemplo abaixo, mostrando os registros que compõem o conjunto de treino $\mathcal{D}$ e sua estrutura. O conjunto é composto por $|\mathcal{D}|$ registros, que pertencem a j consultas, e cada consulta tem um subconjunto disjunto $\mathcal{D}$. Um mesmo documento pode aparecer em mais de uma consulta, mas, nesse caso, lhes são dados identificadores diferentes. Neste exemplo, a consulta de *id* 1 tem a registros no total, e a última consulta começa no b-ésimo registro e termina no último registro $|\mathcal{D}|$.

$$
\begin{aligned}
q:1,\ d_1 \ &: [PageRank(d_1),\ tf(q_t,d_1),\ idf(q_t),\ ...,\ f_m],\ r=1 \\
&\cdots \\
q:1,\ d_a \ &: [PageRank(d_a),\ tf(q_t,d_a),\ idf(q_t),\ ...,\ f_m],\ r=0 \\
&\cdots \\
q:j,\ d_b \ &: [PageRank(d_b),\ tf(q_t,d_b),\ idf(q_t),\ ...,\ f_m],\ r=0 \\
&\cdots \\
q:j,\ d_{|\mathcal{D}|} &: [PageRank(d_{|\mathcal{D}|}),\ tf(q_t,d_{|\mathcal{D}|}),\ idf(q_t),\ ...,\ f_m],\ r=0
\end{aligned}
$$

3.3.1 Learning to Rank com Regras de Associação

Em [38], a combinação de *Learning to Rank* com regras de associação, chamado LRAR *Learning to Rank with Association Rules*, criou um método capaz de providenciar um ranqueamento flexível para novos registros em tempo de consulta (query). O método é flexível pois as regras são criadas a partir de cada novo documento, permitindo uma solução sob demanda que produz apenas as regras de associação necessárias para criar um função de ranqueamento específica. Este método constrói modelos aprendidos de forma *lazy* (preguiçosa).

A função ϕ no LRAR é criada usando um conjunto de regras de associação que são da forma $\mathcal{X} \xrightarrow{\theta} r_i$, onde $\mathcal{X}$ (também chamado de $itemset$) é uma combinação de várias $features$ do conjunto de treino. Por exemplo, $\mathcal{X}_j^z = PageRank(d_j) \wedge tf(q_t,d_j) \wedge idf(q_t)$ refere-se a regra z, documento j, e $features$ compostas pelas funções *PageRank*, tf e idf em relação a d_j [2]. Além disso, r_i é o valor de relevância e θ é a confiança da regra (Seção 3.1.2), ou seja, a chance de $\mathcal{X}$ (criado com os novos registros) aparecer nos documentos do conjunto de treino que têm o valor de relevância r_i. Cada regra também tem um suporte, indicando a frequência de ambos $\mathcal{X}$ e r_i em $\mathcal{D}$.

No LRAR, as regras de associação são usadas para estimar a relevância do documento. O suporte da regra é referido como $\sigma(\mathcal{X} \longrightarrow r_i)$, e a confiança como $\theta(\mathcal{X} \longrightarrow r_i)$. A regra é válida quando os limiares σ_{min} e θ_{min} são satisfeitos. O Algoritmo 5 descreve o LRAR de [38], mas com uma notação simplificada. Cada documento d é processado iterativamente, usando apenas o conjunto de treino projetado $\mathcal{U}_d$ na linha 2, isto é, apenas

[2]Para simplificar a notação, omitimos o número do documento e da regra de $\mathcal{X}$ quando necessário.

Algorithm 5 LRAR - Learning to Rank with Association Rules.

Require: The Reduced Training Data $\mathcal{U}$ and $\mathcal{T}$, thresholds σ_{min} and θ_{min}, Max rule size k
Ensure: A rank value ϕ for each document $d \in \mathcal{T}$

Let $\mathcal{U}_d$ be the set of documents from the projection of document d on a training set $\mathcal{U}$
1: **for all** pair $(d,q) \in \mathcal{T}$ **do**
2: $\mathcal{R}_d \Longleftarrow$ rules of size up to k extracted from $\mathcal{U}_d \mid \sigma \geq \sigma_{min}, \theta \geq \theta_{min}$
3: **for all** $i \mid 0 \leq i \leq c$ **do**
4: $s(d,r_i) = \frac{\sum_{(\mathcal{X} \to r_i) \in \mathcal{R}_d} \theta(\mathcal{X} \to r_i)}{\log |\mathcal{R}_d|}$
5: $\phi \Longleftarrow \sum_{i=0}^{c} \left(r_i \times \frac{s(d,r_i)}{\sum_{j=0}^{c} s(d,r_j)} \right)$

os documentos que compartilham features com d para extrair o conjunto projetado de regras $\mathcal{R}_d$. Um conjunto projetado de regras têm as regras obtidas do documento de teste que existem nos documentos de treino, e restritas a um valor de relevância r_i. Na linha 4 a pontuação (*score*) da relação entre documento e um valor de relevância é obtido. Como muitas regras podem ser geradas quando combina-se três ou mais *features* no antecedente, a função *log* é usada para suavizar a relação entre quantidade e confiança das regras.

A função ϕ do LRAR ,definida na linha 5, é uma combinação linear dos *scores* normalizadas de cada valor de relevância. Depois que cada documento d_j tem sua relevância estimada, o ranqueamento final será de acordo com a ordem decrescente de $\phi(d_j)$.

Como documentos podem ter *features* em comum, um mecanismo de *caching* foi utilizado para guardar regras de associação. Veloso et al. [38] define essa *cache* como um contêiner de entradas na forma $< chave, \; dados >$, onde $chave = \{X, r_i\}$ e $dados = \{\sigma(X \longrightarrow r_i), \theta(X \longrightarrow r_i)\}$. Antes de gerar uma regra, ela é buscada na *cache*. Se encontrada, é utilizada, senão é gerada e inserida na *cache*.

O algoritmo LRAR mostrou ter uma qualidade de ranqueamento relevante contra outros algoritmos de L2R, como foi descrito nos resultados de [38]. Os bons resultados também foram confirmados em [33], onde o SSARP foi aplicado para reduzir o conjunto de treino e melhorar o tempo de processamento, sem prejudicar a qualidade do ranqueamento.

É importante notar que para trabalhar com regras de associação nos conjuntos de dados de *learning to rank*, é necessário que as *features* sejam normalizadas e discretizadas. O método utilizado por Silva et. al no SSARP foi o TUBE (*Tree-based Unsupervised Bin Estimator*) [31], onde o número de *bins* escolhido foi 10 [33]. Assim, o número de itens ou *features* depois da discretização, é aproximadamente 10 vezes maior que o número de *features* do conjunto de dados. Os itens passam a ter uma forma como

tf[0,15-0,25], por exemplo.

3.3.2 Parallel Learning to Rank with Association Rules

PLRAR (*Parallel Learning to Rank with Association Rules*) é a versão paralela do LRAR proposta por Sousa et. al [10], implementada para utilizar as milhares de *threads* que uma GPU oferece. Foi utilizado a coleção LETOR para fazer seu *benchmark* e obteve-se *speedup* médio de 127x no tempo de processamento de uma *query* em relação ao LRAR.

A metodologia do método PLRAR é composta por 3 etapas para viabilizar o L2R sob demanda. A primeira etapa consiste em se obter a base de treino reduzida utilizando o SSARP, que permite gerar menos regras em $\mathcal{R}_d$ para cada documento de teste, sem perder acurácia. Para a coleção LETOR, o conjunto de treino foi reduzido para 1,1% a 2,3% de seu tamanho original. Uma comparação entre o tempo de processamento de uma *query* com o treino original e o reduzido, mostrou que o *speedup* médio do reduzido foi de 124x.

A segunda etapa é criar na CPU uma representação vertical (lista invertida) das *features* que o conjunto de treino possui, e então transferir para a GPU. Para reduzir a transferência de dados da CPU para a GPU, essa representação vertical é feita com mapas de bits (*bitmaps*) usando vetor de inteiros sem sinal, para ser uma representação menor em memória. Cada *feature* tem seu próprio vetor *bitmap*, onde cada *bit* representa se ela ocorre ou não no documento representado por esse *bit*.

A terceira etapa é a execução do PLRAR, onde cada tripla $\langle q, d, ? \rangle$ do conjunto de teste é processada por uma única *thread* da GPU. Como cada *thread* na GPU pode ter um índice único, calculado com as variáveis de índice de *thread* e de bloco, esse índice é associado ao índice do documento.

O Algoritmo 6 descreve o PLRAR. Ele recebe como entrada um tamanho máximo das regras que serão geradas, começando de 1 até `MAX_RULE_SIZE`, onde uma regra de tamanho 2 corresponde a uma regra com somente uma *feature* como antecedente e o nível de relevância como consequente. O algoritmo recebe também um limiar de suporte e confiança, que juntamente com `MAX_RULE_SIZE`, são especificados pelo usuário. O conjunto vertical ou invertido é calculado e o conjunto de teste é lido, ambos pela CPU, e são copiados para a GPU.

Este algoritmo é executado para cada documento i na *thread* i. Na linha 1 é feita a iteração para o tamanho de regra especificado. Na linha 2 é escolhido uma *feature* do documento d_i, onde na linha 3 será utilizada para gerar combinações de um tamanho de acordo com a iteração da linha 1. Nas linhas 4-6, para cada nível de relevância é calculado o suporte e confiança da regra, onde o seu antecedente é a combinação gerada na linha 3, e

o consequente é o nível de relevância. Por último, na linha 11 há o cálculo das pontuações e ranqueamento do documento d_i.

Algorithm 6 PLRAR - Parallel Learning to Rank with Association Rules.

Require: The Reduced Training Data $\mathcal{U}$ and $\mathcal{T}$, thresholds σ_{min} and θ_{min}, Max rule size k
Ensure: A rank value ϕ for each document $d \in \mathcal{T}$

{The next lines of code process the document i by the thread i}
1: **for** $k = 1$ to MAX_RULE_SIZE **do**
2: **for all** feature f in document d_i **do**
3: **for all** combination j with size k that includes f **do**
4: **for all** relevance $rel \in \{r_0, ..., r_c\}$ **do**
5: compute support for rule: $j \longrightarrow rel$
6: compute confidence for rule: $j \longrightarrow rel$
7: compute the ranking for the d_i from all generated rules

O suporte e confiança da regra são calculados fazendo a leitura e processamento dos *bitmaps* das *features* que a compõem. Para encontrar a frequência conjunta (suporte) de duas *features*, uma operação *bit* a *bit AND* é feita entre seus *bitmaps*, e então a quantidade de *bits* 1 é contada no *bitmap* resultante. De Sousa et al. [10] apresentam um exemplo para mais de duas *features*. Sejam f_1, f_3 e f_4 *features* em uma base de treino de 16 documentos, representadas por um *bitmap* de 2 inteiros de 8 *bits* como na Figura 3.6. Para calcular a frequência das 3 *features* conjuntas $f_1 \wedge f_3 \wedge f_4$, primeiro é feito o *AND* de $f_1 \wedge f_3$ e esse resultado intermediário é guardado em uma variável f_x, e então é feito o *AND* de $f_x \wedge f_4$ para dar o *bitmap* final. Conta-se a quantidade de *bits* 1 no *bitmap* usando uma instrução *popcount*, e essa é a frequência para calcular o suporte e a confiança, como mostra a coluna *Count*.

	Document 1 up to 16		**Count**
f_3	1 1 0 1 0 0 1 0	0 1 1 0 0 0 1 0	7
f_1	0 1 0 0 1 0 1 0	0 1 0 0 1 0 1 0	6
f_x	0 1 0 0 0 0 1 0	0 1 0 0 0 0 1 0	4
f_4	0 0 1 0 0 0 1 0	1 0 0 0 1 0 1 0	5
$f_1 \wedge f_3 \wedge f_4$	0 0 0 0 0 0 1 0	0 0 0 0 0 0 1 0	2

Figura 3.6: *Exemplo de interseção de mapas de bits, quando o número de documentos é 16 [10].*

3.3.3 Outros Trabalhos Relacionados

Os trabalhos [9, 41] buscam otimizar métricas de acurácia para gerar um bom modelo de ranqueamento. De Almeida et al. [9] utilizaram programação genética para

gerar funções de ranqueamento que vão evoluindo em cada geração. As funções são armazenadas como árvores, e para aumentar a efetividade do método, essas funções podem ser compostas por outras já conhecidas na literatura, como a TF-IDF (*Term Frequency - Inverse Document Frequency*) [29]. Isto acelerou a convergência para bons resultados e reduziu o *overfitting*. No trabalho de Yue et al. [41] foram utilizadas *support vector machines* (SVM) estruturais para trabalhar com a otimização de MAP (*Mean Average Precision*) [22]. Foi observado que vários outros trabalhos otimizavam para área ROC (*Receiver Operating Characteristic Curve*) ou NDCG (*Normalized Discounted Cumulative Gain*), e isso dava um MAP não ótimo. Sua abordagem direta para o MAP gerou um resultado de maior qualidade, e ainda evitou passos intermediários e o uso de heurísticas, o que melhorou a eficiência também.

Chapelle et al. [8] faz um levantamento de várias questões e situações no *learning to rank* que ainda não foram completamente exploradas. São considerados 8 pontos: o primeiro é sobre a generalização, de como manter a qualidade quando se tem um número muito maior de teste do que treino; o segundo é como manter a acurácia num sistema online, onde pode ser necessário reaprender um modelo, que pode aumentar de complexidade; o terceiro é de como determinar se há tendências nas amostras selecionadas de treino que não costumam refletir bem as de teste. Utilizar ponderação no treino ou selecionar amostras pouco relevantes na sua criação ajudam a corrigir esse problema. Isto é a base do aprendizado ativo; o quarto é sobre o *learning to rank* em larga escala, pois muitos algoritmos preocupam-se somente com a acurácia e esquecem da eficiência e escalabilidade. As soluções geralmente adotadas são o uso de paralelismo, ensembles ou aproximação; o quinto ponto é sobre a robustez, de como as funções devem manter resultados aceitáveis quando expostas a novas *features* e amostras; o sexto ponto é sobre a diversidade do treino, que não deve ser criado baseado somente na relevância; O sétimo ponto é sobre a transferência de aprendizado, de como poder utilizar os dados e modelos de um problema em outro similar, evitando processá-los do zero; O último ponto é sobre *learning to rank* online, de como é um desafio manter uma lista ranqueada e atualizada com documentos de acordo com sua relevância e idade, e manter esse modelo estável e robusto durante as frequentes atualizações.

3.4 k-Vizinhos Mais Próximos

O kNN (k-Nearest Neighbors) ou k-vizinhos mais próximos é um problema de otimização para encontrar os k pontos mais próximos em um espaço métrico. Uma definição formal desse problema é: dado um conjunto $\mathcal{X} = \{x_1, x_2, \ldots, x_n\}$ de n pontos em um espaço $\mathbb{R}^d$ e um ponto de consulta (*query*) q também nesse espaço, os k pontos de $\mathcal{X}$ mais próximos de q devem ser retornados [29]. Essa proximidade depende da métrica

de distância utilizada, que pode ser a distância Euclidiana, de Manhattan, Cosseno ou qualquer outra que for apropriada para o contexto.

O método convencional de busca kNN consiste em calcular a distância de q para todos pontos de $\mathcal{X}$, que é a forma força bruta. Em altas dimensões e com alto número de pontos, essa forma leva um tempo muito alto para fazer a consulta, além de consumir muito espaço computacionalmente. O método eficiente consiste em criar um índice invertido, onde cada dimensão é organizada juntamente com os pontos diferentes de zero de $\mathcal{X}$, permitindo que a consulta seja feita com somente aqueles pontos que irão dar uma distância diferente de zero.

Canuto et al. [7] usaram o kNN para gerar $metafeatures$ que aprimoram a classificação de documentos. Este processo invoca o kNN várias vezes, e é um processo custoso. As $metafeatures$ capturam a informação local e global da possibilidade de um documento pertencer a uma certa classe. Sua implementação, chamada GT-kNN (GPU-based Textual k-Nearest Neighbors), utiliza índice invertido, e permitiu que aplicasse a geração de $metafeatures$ em bases de dados de alta dimensionalidade e alta esparsidade, onde outras soluções nem conseguiam atender os requisitos de memória. Mostraremos a seguir como é criado esse índice invertido.

3.4.1 Indexação dos Dados

O Algoritmo 7 descreve como é feita a indexação dos dados. Esse algoritmo requer que os documentos de entrada e o índice invertido caibam na memória, e que sejam estáticos. Na entrada $\mathcal{E}$, cada documento é organizado em pares de termo e documento (t,d), onde $t \in \mathcal{V}$, e $d \in \mathbb{D}_{in}$, onde $\mathcal{V}$ é o vocabulário representando cada termo único, e $\mathbb{D}_{in}$ os índices dos documentos. Depois que $\mathcal{E}$ é enviado para a GPU, é feita uma contagem em paralelo para preencher o vetor de frequência de documento df ($document\ frequency$), a quantidade de vezes que cada termo aparece. Então é feita uma soma de prefixos exclusiva paralela, usando a biblioteca CUDPP [32], no vetor df na linha 6, onde o resultado é armazenado em $index$. Assim, cada posição de $index$ terá a soma de todos elementos antes dele, onde será utilizado para indicar onde a lista invertida de cada termo começa no índice invertido. Nas linhas 7-10 o índice invertido é preenchido em paralelo, onde cada entrada é o documento junto com o cálculo de seu TF-IDF. Ao mesmo tempo, as normas de cada documento também são calculadas para que possam ser utilizadas no cálculo da distância Cosseno ou Euclidiana.

A Figura 3.7 ilustra os passos de criação do índice invertido com uma entrada de 5 documentos e vocabulário de 5 termos. Para o termo t_3, por exemplo, o vetor $index$ aponta para onde a sua lista invertida $(d_1,\ d_5)$ começa no índice invertido, a posição 5, e termina na posição 6 de acordo com o próximo valor (7 menos 1).

Algorithm 7 DataIndexing($\mathcal{E}$)

Require: term-document pairs in $\mathcal{E}[0..|\mathcal{E}|-1]$
Ensure: df, *index*, *norms*, *invertedIndex*

1: array of integers $df[0..|\mathcal{V}|-1]$ `// document-frequency array, initialized with zeros.`
2: array of integers $index[0..|\mathcal{V}|-1]$.
3: array of floats $norms[0..|\mathbb{D}_{in}|-1]$.
4: $invertedIndex[0..|\mathcal{E}|-1]$ `// the inverted index`
5: Count the occurrences of each term in parallel on the input and accumulates in df.
6: Perform an exclusive parallel prefix sum on df and stores the result in *index*.
7: Access in parallel the pairs in $\mathcal{E}$, with each processor performing the following tasks:
8: Compute the tf-idf value of each pair.
9: Accumulate the square of the tf-idf value of a pair (t,d) in $norms[d]$.
10: Store in *invertedIndex* the entries corresponding to pairs in E, according to *index*.
11: Compute in parallel the square root of the values in array *norms*.
12: **Return** the arrays: df, *index*, *norms* and *invertedIndex*.

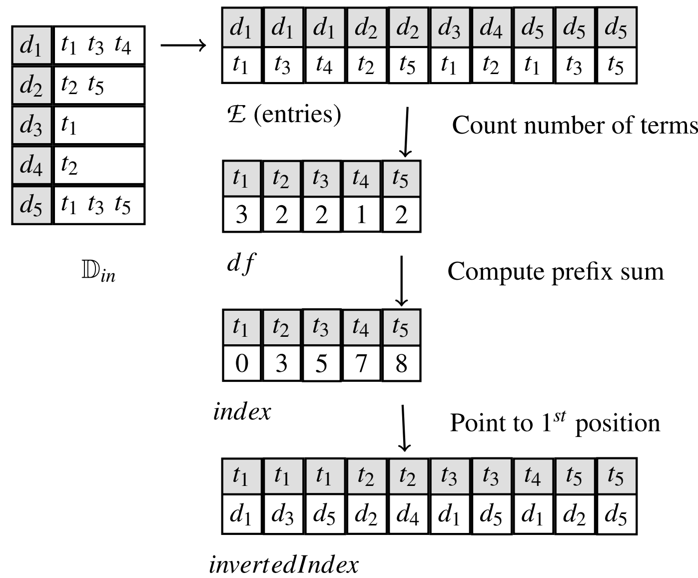

Figura 3.7: *Criação do índice invertido [7].*

3.4.2 Busca dos k-Vizinhos Mais Próximos

A Figura 3.8 ilustra os passos principais do processamento de uma consulta. Neste exemplo, a consulta (*query*) contém três termos, t_1, t_3 e t_4, e a coleção da Figura 3.7 é usada. As estruturas relacionadas à consulta tem a notação "$_q$"como identificador. Primeiro os vetores df_q e $start_q$ são obtidos em paralelo, fazendo uma cópia dos campos correspondentes de cada termo da consulta dos vetores df e *index* (criados durante a indexação). Então uma soma de prefixo inclusiva é feita no vetor df_q, e armazenada em $index_q$. A figura mostra como é feito o mapeamento de cada posição do vetor lógico E_q para as entradas correspondente do vetor do índice invertido, onde é criado um balanceamento de carga entre todos processadores.

Com esse mapeamento, cada entrada computa a distância para os documentos

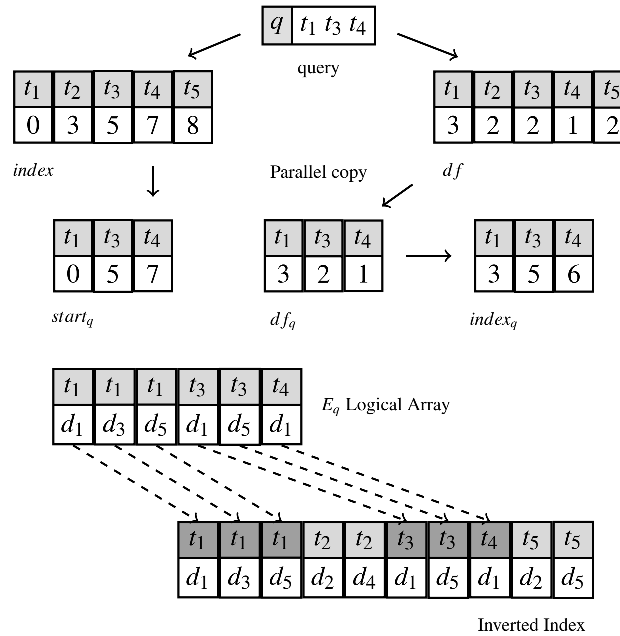

Figura 3.8: *Processamento de uma consulta contendo três termos [7].*

apontados, para que os k vizinhos mais próximos possam ser selecionados, utilizando um algoritmo de ordenação parcial sobre um vetor contendo as distâncias, com o tamanho de $|\mathbb{D}_{in}|$. Para isto, foi implementado uma versão paralela do Truncated Bitonic Sort (TBiS) 'Ordenação Bitônica Truncada', que segundo [35], é um algoritmo superior nesse contexto de kNN sobre outros algoritmos de ordenação parcial. O TBiS permite que os blocos de GPU tenham independência de dados, tirando a necessidade de sincronização entre os blocos. Na implementação do GT-kNN, o TBiS foi implementado com uma estratégia de redução, onde os k mais próximos locais de cada bloco de GPU são copiados em sequência para um único vetor. Esse vetor de tamanho k x número de blocos da GPU é retornado para a CPU, onde uma estrutura de *heap* termina a ordenação.

O Algoritmo 8 mostra como é feita a busca kNN. Nas linhas 4-7 os vetores da consulta df_q e $start_q$ são preenchidos em paralelo. Na linha 8 é feita uma soma de prefixo inclusiva do df_q e armazenada em $index_q$. Nas linhas a seguir, cada processador faz um mapeamento de cada posição x no intervalo de índices de E_q, associado com a posição apropriada no índice invertido. Este mapeamento é descrito nas linhas 10-17 do algoritmo, onde cada processador obtém o deslocamento (offset) e termo correto para manter o balanceamento de carga. Esta entrada mapeada é utilizada para calcular as distâncias entre cada documento das entradas do índice invertido e as da consulta. Por fim, na linha 21 é feita a ordenação parcial com bitonic sort truncado, onde são criadas partições ordenadas, que serão mescladas na CPU em uma estrutura *heap* para obter os k documentos mais próximos.

Algorithm 8 kNNSearch(invertedIndex, q)

Require: $invertedIndex$, df, $index$, $query$ $q[0..|\mathcal{V}_q|-1]$
Ensure: distance array $dist[0..|\mathbb{D}_{in}|-1]$ initialized according to the distance function used

1: array of integers $df_q[0..|\mathcal{V}_q|-1]$ initialized with zeros
2: array of integers $index_q[0..|\mathcal{V}_q|-1]$
3: array of integers $start_q[0..|\mathcal{V}_q|-1]$
4: **for** each term $t_i \in q$, in parallel **do**
5: $df_q[i] = df[t_i]$;
6: $start_q[i] = index[t_i]$;
7: Perform an inclusive parallel prefix sum on df_q and stores the results in $index_q$
8: **for** each processor $p_i \in \mathcal{P}$ **do**
9: **for** $x \in [i\lceil\frac{|E_q|}{|\mathcal{P}|}\rceil, min((i+1)\lceil\frac{|E_q|}{|\mathcal{P}|}\rceil-1, |E_q|-1)]$ **do**
10: // Map position x to the correct position of the $invertedIndex$
11: $pos = min(i : index_q[i] > x)$;
12: **if** $pos = 0$ **then**
13: $p = 0$; $offset = x$
14: **else**
15: $p = index_q[pos-1]$; $offset = x-p$;
16: $indInvPos = start_q[pos] + offset$
17: uses $q[pos]$ and $invertedIndex[indInvPos]$ in the partial computation of the distance between q and the document associated to $invertedIndex[indInvPos]$
18: Perform a partial sort on the distances using a truncated bitonic sort and a reduction operation
19: **Return** the array: $dist$ with partially sorted partitions.

3.4.3 Algoritmo GPU-based Textual k-Nearest Neighbors (GT-kNN)

O Algoritmo 9 descreve a parte principal do GT-kNN. Primeiro o índice invertido é criado, chamando o Algoritmo 7. Então, cada consulta q é lida em um vetor, e é feita a alocação na GPU das variáveis e estruturas auxiliares para a busca, nas linhas 2-4. Nas linhas 5-6, o Algoritmo 8 é chamado para retornar a lista parcialmente ordenada, e a lista é finalmente ordenada em uma estrutura *heap* na CPU. Esta alocação de memória para cada consulta tem um pequeno impacto no tempo de processamento, conforme veremos na Seção 5.4.

Algorithm 9 GPU-based Textual-kNN($\mathcal{E}$, $\mathcal{Q}$)

Require: term-document pairs in $E[0..|\mathcal{E}|-1]$, list of queries (documents) $\mathcal{Q}$
Ensure: A list of k nearest neighbors, one for each query.

1: $invertedIndex = DataIndexing(E)$;
2: **for** each query $q \in \mathcal{Q}$ **do**
3: Read q and prepare its array;
4: Allocate memory in GPU for q and its auxiliary structures;
5: $partialList = kNNSearch(invertedIndex, q)$;
6: Output k elements after sorting $partialList$ in a heap structure
7: Free the GPU memory for q

3.4.4 Outros Trabalhos Relacionados

Garcia et al. [12] aceleraram o algoritmo kNN (*k Nearest Neighbors*) por força bruta usando GPU. O kNN possui várias aplicações, mas sua complexidade computacio-

nal cresce polinomialmente com o tamanho dos dados. Em sua versão de GPU, o espaço de dimensões pouco afeta o tempo de execução, e permite trabalhar com mais dados e em tempo razoável. A implementação em GPU consegue atingir um speedup de até 120x em relação a uma versão sequencial em CPU, e 40x ao método com árvore KD (k-dimensional).

Lukač e Žalik [21] trabalharam com o problema do kNN quando se tem muitas dimensões, e criam uma solução aproximada para o algoritmo usando *locality–sensitive hashing* (LSH) [13]. A paralelização usa um variante do LSH com *multi–probe* para acelerar a consulta de todos pontos, e uma *skip–list* é usada para acelerar a atualização dos resultados. Os pontos são distribuídos em baldes, e a consulta é feita nos baldes ao lado. Esta abordagem ajudou a ter uma chance de 90% de encontrar os vizinhos reais. O *speedup* obtido foi de 30x em relação a versão sequencial. Kato e Hosino [19] conseguiram trabalhar com *datasets* maiores usando múltiplas GPUs e particionando os dados em blocos de uma abordagem que só precisa da matriz triangular superior. Mas um pedaço de bloco pode não caber totalmente quando a dimensão é muito grande. Com duas GPUs conseguiram um *speedup* de 330x em relação à versão serial.

Nos trabalhos [30, 7] um índice invertido foi criado para compactar a estrutura esparsa e de alta dimensionalidade de bases textuais. Esta abordagem diminuiu o consumo de memória em pelo menos 4000 vezes, e permitiu trabalhar com *datasets* grandes (RCV1 e Medline) e com uma solução exata, ao contrário de outros trabalhos [21, 12]. Rocha et al. [30] usaram similaridade cosseno e $k = 30$ em seu experimento, e obtiveram um *speedup* de 3x na GPU em relação a versão *multicore* com 6 *threads*.

Zhou et al. [44] também criam um índice invertido, mas com uso genérico, capaz de trabalhar com diferentes tipos de dados complexos. As estruturas complexas são quebradas em partes menores usando um processo chamado *Shotgun and Assembly*, onde o índice invertido é criado usando estas partes. Seu método é aproximado e usa o LSH para fazer múltiplas consultas em paralelo.

CAPÍTULO 4

Algoritmos e Implementações Propostas

Neste capítulo discutimos sobre nossos algoritmos paralelos propostos, o PSSARP na Seção 4.1, PRSS na Seção 4.2, PROFL na Seção 4.3, MT-kNN na Seção 4.4 e FiSH-kNN na Seção 4.5. Os algoritmos implementados com OpenMP usaram as diretivas de compilação `#pragma omp parallel` e `#pragma omp parallel for`.

4.1 Parallel SSARP (PSSARP)

Nesta seção apresentamos nossa solução *multicore* do SSAR, o PSSAR, e consequentemente do SSARP, para servir de *baseline* para nossa proposta *manycore* PRSS.

A primeira parte para viabilizar o ranqueamento sob demanda, é a criação de uma base de treino reduzida que permite acelerar a criação dos modelos. Então é importante que essa primeira fase também seja rápida para dar início ao processo de ranqueamento.

4.1.1 Os Algoritmos PSSAR e PSSARP

No Algoritmo 10 temos nossa proposta de paralelização do SSAR (Seção 3.2.1). A parte de maior peso é a geração de regras de associação, que tem complexidade combinatorial, e é feita para todos documentos a cada iteração. Então propomos paralelizar o processamento dos documentos, onde cada processador trabalha em um subconjunto desses documentos. Isso corresponde à linha 4 do algoritmo, onde o restante é igual ao Algoritmo 3 (SSAR [33]). A implementação foi feita com o uso de OpenMP,

Como o SSARP (Algoritmo 4) aplica o SSAR em cada partição, o Algoritmo 11 PSSARP (Parallel SSARP) é o SSARP que utiliza o PSSAR no lugar do SSAR na linha 8. Uma abordagem por partição, onde cada processador trabalha em uma partição, também é possível. Mas como não é possível saber quando uma partição irá convergir, alguns processadores poderão ficar ociosos.

Este algoritmo também representa a proposta *manycore* PRSS, com cada processador (*thread*) processando um documento, que veremos na Seção 4.2.

Algorithm 10 PSSAR - Parallel Selective Sampling using Association Rules.

Require: The training data $\mathcal{D}$
Ensure: The reduced training data $\mathcal{U}$
Let $\mathcal{W}_{y_i}$ be the set of documents from the projection of document y_i on a training set $\mathcal{W}$
1: $y \Longleftarrow d_i$ such that $\forall d_j : |\mathcal{D}_{d_i}| \geq |\mathcal{D}_{d_j}|$
2: append y to $\mathcal{U}$
3: **loop**
4: **for all** document $d_i \in \mathcal{D}$ in **parallel do**
5: $\mathcal{R}_{d_i} \Longleftarrow$ rules extracted from $\mathcal{U}_{d_i}$
...

Algorithm 11 PSSARP - Parallel Selective Sampling using Association Rules w/ Partitions.

Require: The training data $\mathcal{D}$, Q partitions, Ordered features by χ^2 distribution $\mathcal{F}$
Ensure: The reduced training data $\mathcal{U}$
1: $\mathcal{U} = \emptyset$
2: **for all** $q \mid 0 \leq q < Q$ **do**
3: ...
8: $\mathcal{U} = \mathcal{U} \cup PSSAR(\mathcal{D}^q)$

4.1.2 Ilustração do PSSAR

A Figura 4.1 ilustra de modo geral as implementações *multicore* e *manycore* do PSSAR, seguindo a Figura 3.4 do SSAR. Considerando T *threads*, cada uma fica responsável por calcular as regras de um documento, fazendo as combinações das *features* e suas interseções de *bitmaps*. No *multicore* cada *thread* processará um subconjunto de documentos, e no *manycore* apenas um documento.

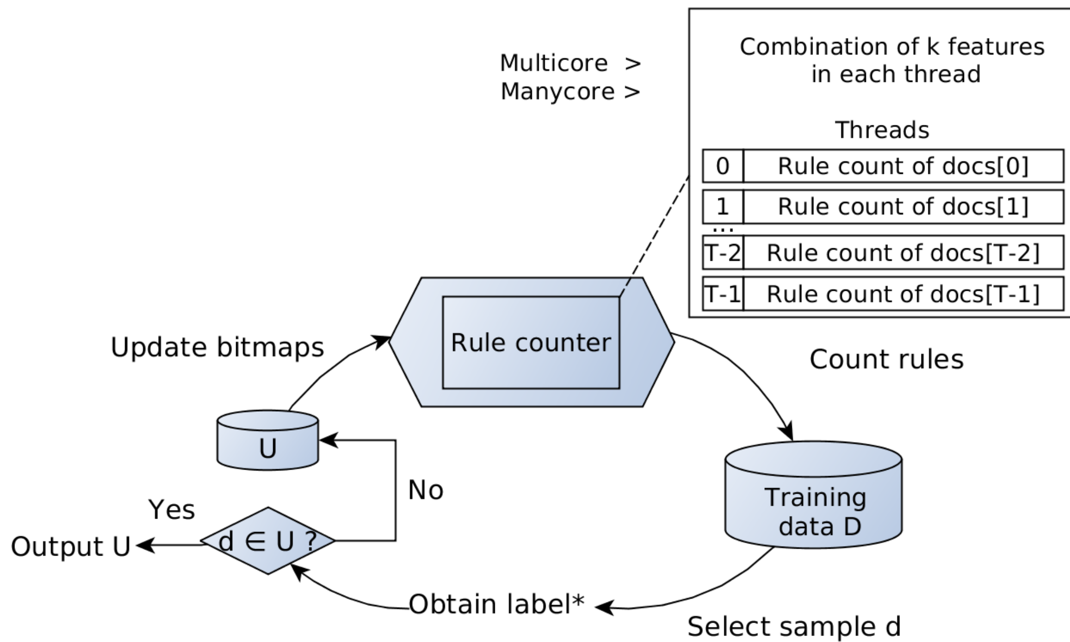

Figura 4.1: *Visão geral das implementações multicore e manycore do PSSAR.*

4.2 Parallel Rule-based Selective Sampling (PRSS)

Nesta seção apresentamos nossa proposta Parallel Rule-based Selective Sampling (PRSS), uma versão *manycore* do algoritmo SSARP (Seção 3.2.1) para acelerar o processo de redução de conjunto de dados. Discutiremos primeiro as estruturas de dados e técnicas utilizadas. Então descrevemos o algoritmo PRSS com uma implementação para uma e múltiplas GPUs.

4.2.1 Representação dos Dados e Abordagem por Thread

O algoritmo SSARP usa regras de associação para selecionar documentos ativamente do conjunto de treino inteiro, baseado em quantas regras foram geradas para cada documento. Para poder trabalhar com conjuntos de dados grandes, a representação dos dados é um fator crítico, pois isso define o padrão de acesso à memória (especialmente em GPUs). Por isso, usamos a mesma representação com *bitmaps* do PLRAR (Seção 3.3.2) para representar listas invertidas, pois permite o cálculo rápido de interseções com o uso de operações bit a bit *AND* (`&`). Além disso, o cálculo de suporte é feito com a instrução `popcount` do CUDA, e os *bitmaps* das classes de relevância são armazenados na memória compartilhada (`__shared__`). Como o acesso à essa memória é mais rápido, isso faz com que as interseções finais sejam mais rápidas também.

No PRSS a operação com *bitmap* foi melhorada ao implementar os elementos como tipos vetorizados de inteiros (`int4`), que força o compilador a produzir instruções de *load* e *store* de 128 *bytes*. Assim, há um aumento na vazão de memória da GPU [1].

O PRSS usa uma abordagem de um documento por *thread*, porque trabalha com todo o conjunto de treino. Nesta situação o PRSS está trabalhando com muitas consultas, onde cada uma costuma ter muitos documentos. Assim, uma abordagem de documento por *thread* é suficiente para garantir uma boa ocupação da GPU. Outra razão para essa escolha é devido ao algoritmo SSARP trabalhar com um menor conjunto de *features* por vez, fazendo o custo de processamento de um documento ficar menor.

4.2.2 O Algoritmo PRSS

O Algoritmo 12 descreve o PRSS, onde uma partição é processada por vez. As entradas são as mesmas que a do algoritmo SSARP. Na linha 1, é declarada a variável onde as partições reduzidas serão armazenadas, e o número de documentos é atribuído com o número de registros do conjunto não-rotulado. Na linha 2, um vetor de ocorrências é construído para cada item (*features* discretizada). A linha 3 aloca as seguintes variáveis,

[1] CUDA Pro Tip: Increase Performance with Vectorized Memory Access

respectivamente: a partição, contador de regras para cada documento, *bitmaps* dos itens e o conjunto de itens atuais que existem no conjunto reduzido.

Da linha 4 em diante, cada partição é processada sequencialmente. Na linha 5, N conta o número atual de documentos inseridos, e *rule_count* guarda o número de regras geradas para cada documento. A linha 6 recebe as colunas do conjunto de dados relevantes para a atual partição, e na linha 7 um vetor da soma das ocorrências de cada item que compõem um documento é construído. O documento com a maior soma (maior projeção) é escolhido como o mais representativo, e é inserido no conjunto reduzido nas linha 8-9. Na linha 10 a partição é copiada para a GPU, para que o *kernel* de GPU possa acelerar a geração de regras. Esta etapa de preparação finaliza na linha 11, onde as variáveis de GPU *g_bitmaps* e *g_cur_items* são resetadas.

Algorithm 12 PRSS - Parallel Rule-based Selective Sampling.

Require: Training data $\mathcal{D}$, $\mathcal{Q}$ partitions, Ordered features by χ^2 distribution $\mathcal{F}$
Ensure: The reduced training data $\mathcal{U}$
1: reduced_sets[$\mathcal{Q}$] = $\{\emptyset\}$, n_docs = $|\mathcal{D}|$
2: occurs[] = build_occurrences($\mathcal{D}$)
3: Allocate g_$\mathcal{P}$[], g_rule_cnt[], g_bitmaps[], g_cur_items[]
4: **for all** $q \mid 0 \leq q < \mathcal{Q}$ **do**
5: $N = 0$, rule_count[$|\mathcal{D}|$]
6: $\mathcal{P}$ = partitioner(q, $\mathcal{D}$, $\mathcal{F}$)
7: doc_occurs[] = build_doc_occurs($\mathcal{P}$, occurs)
8: d = most_representative($\mathcal{P}$, doc_occurs)
9: reduced_sets[q].insert(d)
10: Copy $\mathcal{P}$ to g_$\mathcal{P}$
11: Clear g_bitmaps and g_cur_items
12: **loop**
13: ««« update_bitmap_and_cur_items_kernel »»»(g_$\mathcal{P}[d]$, N, g_bitmaps, g_cur_items)
14: $N = N + 1$
15: «««rule_counter_kernel »»» (n_docs, k, σ_{min}, g_$\mathcal{P}$, g_bitmaps, g_cur_items, g_rule_cnt)
16: Copy g_rule_cnt to rule_count
17: $d = i$ such that $\forall j$: rule_count[i] $\leq$ rule_count[j]
18: **if** reduced_sets[q].find(d) **then break**
19: **else** reduced_sets[q].insert(d)
20: $\mathcal{U}$ = merge(reduced_sets, $|\mathcal{Q}|$)

O processo de seleção incremental é descrito nas linhas 12-19. Um *kernel* é lançado na linha 13 para atualizar os *bitmaps* e o conjunto de itens atuais. A contagem das regras é feita no *kernel* principal (*rule_counter_kernel*) na linha 15, que será detalhado no Algoritmo 13. Depois que o *kernel* termina, a CPU copia o vetor resultante da contagem de regras da GPU para o vetor da CPU (linha 16). A CPU seleciona o documento com o menor número de regras geradas na linha 17 [2], e tenta inserir no atual conjunto reduzido. Se o documento já foi inserido, a convergência foi atingida e a próxima partição será processada. Todos conjuntos reduzidos são mesclados na linha 20 no final.

[2]Em caso de empate, o documento com menor valor em *doc_occurs*[] é selecionado. Havendo outro empate, seleciona-se aquele com maior *id*.

Algorithm 13 Rule Counter.

Require: N, Max rule size k, σ_{min}, Partition g_$\mathcal{P}$, g_bitmaps, g_cur_items, g_rule_cnt
Ensure: The count of rules in g_rule_cnt
{The following code is executed by N threads identified by t.}
1: $size = 0$, m[]
2: **for all** $i \mid 0 \leq i < | g_\mathcal{P}[\,t\,] |$ **do**
3: **if** $g_\mathcal{P}[t][i] \in g_cur_items$ **then**
4: $m[size{+}{+}] = g_\mathcal{P}[t][i]$
5: g_rule_cnt[t] = gen_rules(m, $size$, k, σ_{min}, g_bitmaps)

O Algoritmo 13 descreve o *kernel* Rule Counter, responsável pela contagem de regras, mostrando a execução de uma única *thread* t. Nas linhas 1-4, o documento t é projetado, isto é, o atual documento com apenas os itens que existem no atual conjunto g_cur_items, e atribuído em m. Então na linha 5, as regras de tamanho até k são geradas para esse documento, usando os *bitmaps* para fazer as interseções das combinações dos $size$ itens em m como na Figura 3.6.

4.2.3 PRSS com Múltiplas GPUs

Adicionalmente, foi implementado uma versão Multi-GPU do PRSS, descrito no Algoritmo 14. A solução utiliza OpenMP para lançar um número de *threads* de CPU igual ao número de GPUs, e cada GPU recebe um número igual de documentos para processar. As modificações do algoritmo PRSS consiste em alocações e atribuições para cada variável de GPU, cópias com a quantidade e $offset$ apropriados para cada GPU, e o lançamento do Rule Counter *kernel* em cada subconjunto assinalado.

Ao explorar o conjunto reduzido, nossa implementação paralela do algoritmo de L2R pode ser feita na memória da GPU, entre outras vantagens, como descrito na próxima seção.

4.3 Parallel Rule-based On the Fly Learning to Rank (PROFL)

Nesta seção descrevemos o algoritmo PROFL (Parallel Rule-based On the Fly Learning to Rank), nossa proposta paralela do método LRAR 3.3.1. Começamos apresentando as principais técnicas usadas em nossa solução, especificamente a abordagem por bloco de *threads*, os conceitos de combinadic e $fingerprint$, e o sistema de *cache*. A seguir, o algoritmo PROFL é descrito com uma implementação para uma e múltiplas GPUs. Assim como nossa implementação PRSS, o PROFL também usa as listas invertidas em forma de mapas de bits descritas na Seção 4.2.1.

Algorithm 14 Multi-GPU PRSS.

Require: Training data $\mathcal{D}$, $\mathcal{Q}$ partitions, Ordered features by χ^2 distribution $\mathcal{F}$, #GPUs
Ensure: The reduced training data $\mathcal{U}$

1: reduced_sets[$\mathcal{Q}$] $= \{\emptyset\}$, n_docs = $\lceil \frac{|\mathcal{D}|}{\#GPUs} \rceil$
2: occurs[] = build_occurrences($\mathcal{D}$)
3: **for all** $c \mid 0 \leq c < \#GPUs$ in **parallel do**
4: set_gpu_device(c)
5: Allocate g_$\mathcal{P}$[c]]], g_rule_cnt[c][], g_bitmaps[], g_cur_items[c][]
6: **for all** $q \mid 0 \leq q < \mathcal{Q}$ **do**
7: $N = 0$, rule_count[$|\mathcal{D}|$]
8: $\mathcal{P}$ = partitioner(q, $\mathcal{D}$, $\mathcal{F}$)
9: doc_occurs[] = build_doc_occurs($\mathcal{P}$, occurs)
10: d = most_representative($\mathcal{P}$, doc_occurs)
11: reduced_sets[q].insert(d)
12: **for all** $c \mid 0 \leq c < \#GPUs$ in **parallel do**
13: set_gpu_device(c)
14: Copy interval of documents [$\mathcal{P}_{g \cdot n_docs}$, $\mathcal{P}_{min((g+1) \cdot n_docs,\ |\mathcal{D}|)}$] to g_$\mathcal{P}$[$c$]
15: Clear g_bitmaps[c] and g_cur_items[c]
16: **loop**
17: **for all** $c \mid 0 \leq c < \#GPUs$ in **parallel do**
18: set_gpu_device(c)
19: update_bitmap_and_cur_items_kernel (g_$\mathcal{P}$[c][d], N, g_bitmaps[c], g_cur_items[c])
20: $N = N + 1$
21: **for all** $c \mid 0 \leq c < \#GPUs$ in **parallel do**
22: set_gpu_device(c)
23: rule_counter_kernel (n_docs, k, σ_{min}, g_$\mathcal{P}$[c], g_bitmaps[c],g_cur_items[c], g_rule_cnt[c])
24: Copy g_rule_cnt[c] to rule_count[$c \cdot n_docs$]
25: $d = i$ such that $\forall j$: rule_count[i] $\leq$ rule_count[j]
26: **if** reduced_sets[q].find(d) **then break**
27: **else** reduced_sets[q].insert(d)
28: $\mathcal{U}$ = merge(reduced_sets, $|\mathcal{Q}|$)

4.3.1 Abordagem por Bloco de Threads

Na abordagem por bloco de *threads* o paralelismo é explorado com muitos blocos trabalhando em paralelo, onde existem várias *threads* em cada bloco. Com esta estratégia, os documentos de teste (documentos não vistos) podem ser processados em paralelo ao serem atribuídos a um bloco, onde muitas *threads* irão processar um mesmo documento de forma paralela. Uma carga de trabalho balanceada é atingida ao atribuir um número semelhante de regras a serem criadas para cada *thread*. O número de regras para cada *thread* depende do número de itens (*features*) do documento (definido como m), o número de *threads* em um bloco (*num_threads*), e também o valor k, que é um parâmetro definido pelo usuário para indicar o tamanho máximo da regra.

Primeiro atribuímos um índice $z \mid 0 \leq z < \binom{m}{k}$ para cada regra de tamanho k, e então $\left\lceil \frac{\binom{m}{k}}{num_threads} \right\rceil$ regras são atribuídas para cada *thread*. Isso é feito para o tamanho mínimo 2 até k. Escolhemos a abordagem por bloco de *threads* por causa de sua escalabilidade na GPU, e por permitir uma ocupação completa do *hardware*. Como geralmente há centenas ou milhares de documentos associados com uma consulta (*query*), podemos lançar bastante blocos, cada um contendo muitas *threads*. Deste modo, a latência de memória, que é alta em GPUs, pode ser escondida com mais eficiência.

4.3.2 Combinadic

Como citado na Seção 4.3, o combinadic [5] (Chamado recentemente por esse nome) fornece um elemento de um conjunto de combinações distintas de valores inteiros, dado um índice. O combinadic é o ponto chave da nossa proposta de bloco de *threads* e sua escalabilidade. Sua utilização permite que cada *thread* possa obter rapidamente o subconjunto adequado de regras a serem criadas.

O Algoritmo 15 descreve o combinadic, que recebe um valor de índice z, o número de elementos n, e o tamanho da combinação k, onde $0 \leq z < \binom{n}{k}$ [5]. Esse algoritmo fornece um elemento de uma combinação lexicográfica de valores inteiros, que seria um *itemset* (antecedente da regra) no nosso caso. O algoritmo mapeia o valor z para um subconjunto de tamanho k de $\mathcal{X} = \{0, 1, 2, ..., (n-1)\}$, resolvendo a equação $z = \binom{c_1}{k} + \binom{c_2}{k-1} + ... + \binom{c_k}{1}$, restrito a $n > c_1 > c_2 > ... > c_k$. Para isto, o algoritmo busca pelo y no qual o atual binômio $\binom{y}{k}$ é menor ou igual ao atual z, em ordem decrescente de y. Cada y que satisfaz essa inequalidade é adicionado em uma lista, que é retornada no fim do algoritmo.

Algorithm 15 Combinadic – Geração de um subconjunto de valores inteiros.

Require: Index z, Number of elements n, combination size k
Ensure: A lexicographical combination
Let *comb* be a list
1: **for** $y = (n-1)$ **to** 1 **do**
2: **if** $\binom{y}{k} \leq z$ **then**
3: *comb*.add(y)
4: $z = z - \binom{y}{k}$
5: $k = k - 1$
6: **Return** *comb*

Com o combinadic cada *thread* é atribuída com somente um valor de índice para poder obter as regras. Deste modo, cada *thread* pode calcular suas próprias regras independentemente assim que receber o índice, sem precisar calcular as combinações anteriores.

4.3.3 Reaproveitando Itemsets

No PROFL, as regras são criadas a cada novo documento de teste. Considerando a similaridade entre esses documentos, há uma grande demanda para as mesmas regras. Então, uma estratégia de *cache* seria apropriada nesse caso. Em nossa estratégia, antes de uma regra ser criada, a *thread* busca o *itemset* em uma tabela *hash*. Para fazer isto, implementamos a função de *fingerprint* desenvolvida por Atreas e Karanikas [4], que

mapeia um conjunto de valores inteiros, um *itemset*, para um número de ponto flutuante. A Equação 4-1 descreve a função de *fingerprint*.

$$fingerprint(X) = \sum_{i=1}^{n} \frac{f_i}{p_i} \tag{4-1}$$

onde $X = \{f_1, f_2, ..., f_n\}$ e p_i é o i-ésimo número primo maior que o valor máximo de índice para um f_i.

Para poder mapear o resultado da função de *fingerprint* na tabela *hash*, o número de ponto flutuante é transformado ao interpretar seus 64 bits como um inteiro L de 64 bits. Assim, este inteiro pode ser usado em uma função simples de *hash* com operação de módulo e o tamanho da tabela, $L \ \% \ HASH_SIZE$. A tabela *hash* armazena o *bitmap* do *itemset*, e também o suporte para cada valor de relevância. Quando o *itemset* é encontrado, apenas os valores do suporte são retornados, que são necessários para calcular a confiança.

Uma vantagem importante dessa função de *fingerprint* é o processamento rápido para obter o valor de índice de um *itemset*. Então, se um *itemset* não for encontrado, o PROFL busca na tabela *hash* o subconjunto correspondente ao prefixo de tamanho k-1 com seu *fingerprint*. Assim, um novo *itemset* é criado levando em consideração interseções feitas anteriormente.

Nós usamos um vetor para armazenar a tabela *hash*, e endereçamento aberto com espalhamento linear como estratégia de tratamento de colisões. Apesar do uso dessa estratégia de *hashing* diminuir a quantidade de acessos aglutinados à memória de algumas *threads*, o tempo salvo ao criar *itemsets* melhorou o tempo de processamento em geral, como descrito na Seção 5.3.

4.3.4 O Algoritmo PROFL

Nesta subseção descrevemos o algoritmo do PROFL, levando em consideração as explicações sobre a abordagem por bloco de threads, o algoritmo combinadic, tabela *hash* com *fingerprint*, e a representação por *bitmap* como foram descritos nas subseções anteriores.

O PROFL começa na CPU, recebendo como entrada os documentos de treino do conjunto de treino reduzido, os novos documentos de teste, o tamanho máximo da regra, e os limiares de suporte e confiança. Cada documento é representado como uma lista de m *features* discretizadas, que são transformadas em *itemsets* de tamanho 1. A representação por *bitmap* do conjunto de treino é processada na CPU. Depois disso, as posições de memória na GPU são alocadas, e cada documento de teste d_i é processado pelo Algoritmo 16.

Algorithm 16 PROFL – Parallel Rule-based On the Fly Learning to Rank.

```
Require: Training Set Bitmap, Unseen document with n items, max_rule_size, σ_min and θ_min thresholds
Ensure: The ranking score
    {The following code is executed by threads identified by t.}
 1: for k = 1 to max_rule_size do
 2:   #rules = ⌈ (m choose k) / num_threads ⌉
 3:   ruleIndexes = from t ∗ #rules to min((t+1) · #rules, (m choose k))
 4:   for all ruleIndexes z do
 5:     itemset = Combinadic(z, n, k)
 6:     h = fingerprint(itemset)
 7:     if Cache(h) is true then
 8:       p = Cache(h)
 9:     else
10:       L = ∅
11:       for s = (k-1) to 1 do
12:         h = fingerprint( prefix(itemset, s) )
13:         if Cache(h) is true then
14:           include Cache(h) in L
15:           break
16:       p = create_itemset(itemset, L)
17:       insert itemset into cache
18:     for all relevance value rel do
19:       check σ(p ⟶ rel) ≥ σ_min and θ(p ⟶ rel) ≥ θ_min
20:   Sum the confidence value from several rules with a parallel reduction
21: Compute the ranking for document_i from all generated rules
```

O Algoritmo 16 descreve a parte do PROFL que é processada na GPU (*kernel*). Na linha 1 o limite do tamanho do *itemset* é definido. Nas linhas 2-3, o índice disjunto para as regras é criado, e o algoritmo combinadic é chamado na linha 5 para obter o *itemset* da regra. Na linha 6 o algoritmo aplica a função de *fingerprint* para checar se o *itemset* já foi criado. Se encontrado na tabela *hash*, o suporte do *itemset* é retornado na linha 8. Se não, o algoritmo busca por um *itemset* menor na tabela, o prefixo de tamanho s do *itemset* original, e insere esse prefixo em L, nas linhas 10-15. A busca de *itemsets* menores é feita para poder aproveitar alguns *bitmaps* já criados. Na linha 16 um novo *itemset* é criado usando o prefixo L. Nas linhas 18-19 é checado se os valores de suporte e confiança das regras satisfazem os limiares. Se satisfazerem, os valores de confiança de várias regras são somados (linha 20). Para isto, aplicamos uma redução paralela no vetor de memória compartilhada do bloco, como descrito na Seção 2.5. Finalmente, na linha 21 o ranqueamento pode ser computado usando os *scores* recebidos de cada documento.

4.3.5 PROFL com Múltiplas GPUs

Além de nossa proposta do PROFL que usa uma GPU, também fizemos uma versão multi-GPU. O Algoritmo 17 mostra a implementação multi-GPU do PROFL, onde cada GPU é controlada por uma *thread* de CPU. O algoritmo começa com a associação das *threads* de CPU com as GPUs, e copia o *bitmap* do conjunto de treino para as memórias das GPUs. Depois disso, os documentos de teste são divididos igualmente entre

Algorithm 17 Multi-GPU PROFL.

Require: Training Set Bitmap, Unseen Documents, #GPUs, max_rule_size, σ_{min} and θ_{min} threshold
Ensure: The ranking score
1: **for** $g = 0$ **to** #GPUs **parallel do**
2: set_gpu_device(g)
3: Copy training set bitmap to GPU g
4: $\#docs = \lceil \frac{N}{\#GPUs} \rceil$
5: $unseenDocumentSet$ = from $g \cdot \#docs$ to $min((g+1) \cdot \#docs, N)$
6: «««PROFL»»»(Training Set Bitmap, unseenDocumentSet, max_rule_size, σ_{min}, θ_{min})

as GPUs, e cada uma invoca o Algoritmo 16. A paralelização da CPU foi feita usando a API OpenMP.

4.3.6 Ilustração do PROFL

Na Figura 4.2 temos a ilustração da execução do PROFL, mostrando o trabalho de um bloco. Nele os documentos de treino reduzido são transformados em $10 \cdot m$ $bitmaps$, onde m é o tamanho dos documentos e a quantidade original de $features$, e 10 é o valor usado na discretização. São lançados B blocos, que correspondem ao número N de documentos. E então cada id do bloco corresponde ao índice do documento. Todos blocos da GPU tem acesso a uma mesma tabela hash global, que é acessada pelas $threads$ com o uso do $fingerprint$. Com o uso do combinadic, cada uma das T $threads$ do bloco processam uma mesma quantidade de regras $range$, no intervalo de acordo com seu índice. No final, os $scores$ são somados com uma redução paralela e retornados para a CPU, onde o rank é dado para o documento.

Também foi implementada uma versão do PROFL sem o uso do combinadic, seguindo a mesma abordagem de documento por $thread$ do PLRAR (Algoritmo 6). A Figura 4.3 ilustra essa versão. O número de blocos lançados é igual ao número de documentos dividido pelo número de $threads$ por bloco. Cada $thread$ calcula e soma os $scores$ do documento associado à ela, usando o seu id e o do bloco para identificar o documento. E cada $score$ é retornado para a CPU.

4.4 MultiThreaded k-Nearest Neighbors (MT-kNN)

Nesta seção descrevemos a implementação $multicore$ que fizemos do GT-kNN, chamada de MultiThreaded-kNN (MT-kNN), para servir de $baseline$ para nossa proposta *manycore* FiSH-kNN. Utilizamos a API OpenMP para fazer um paralelismo por consulta (documento), onde cada processador faz a busca kNN de um subconjunto de consultas. No MT-kNN, a indexação dos dados é a mesma que a do Algoritmo 7 DataIndexing, mas considerando as $threads$ de CPU.

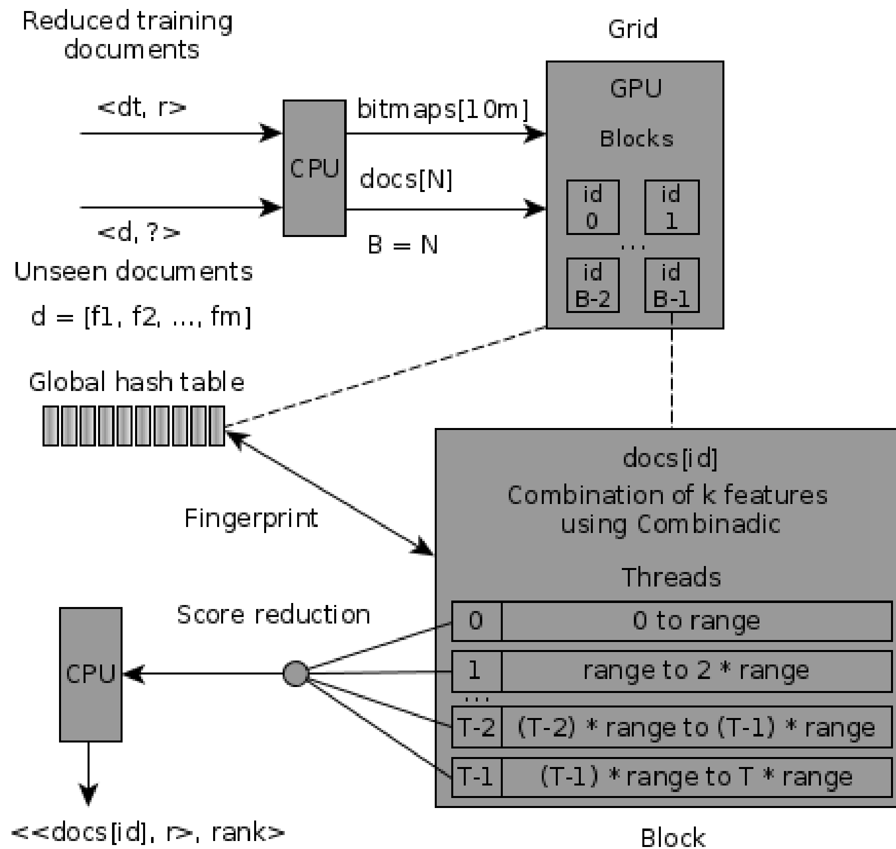

Figura 4.2: *Ilustração do PROFL.*

4.4.1 Busca na CPU

No Algoritmo 18 temos o CPU-kNNSearch, onde uma *thread* de CPU faz a busca kNN. As variáveis são as mesmas que do Algoritmo 8, e cada termo é processado sequencialmente na linha 6. Com bases nos vetores $start_q$ e df_q é possível saber quantas entradas devem ser visitadas no índice invertido para o termo atual, e computar as distâncias nas linhas 7-9. Nesta versão, foi usada a função de ordenação Sort da biblioteca STL do C++, então não há partições ordenadas parcialmente.

4.4.2 O Algoritmo MT-kNN

O Algoritmo 19 mostra o MT-kNN em si. Nele é criado um único índice invertido, onde a indexação é feita em paralelo pelas *threads* de CPU na linha 1. E na linha 2 cada *thread* processa uma consulta por vez em paralelo, chamando o Algoritmo 18. Como o vetor retornado já é ordenado nesse caso, a saída para consulta são os k primeiros elementos desse vetor. Quando executado com apenas uma *thread*, este algoritmo torna-se a versão sequencial.

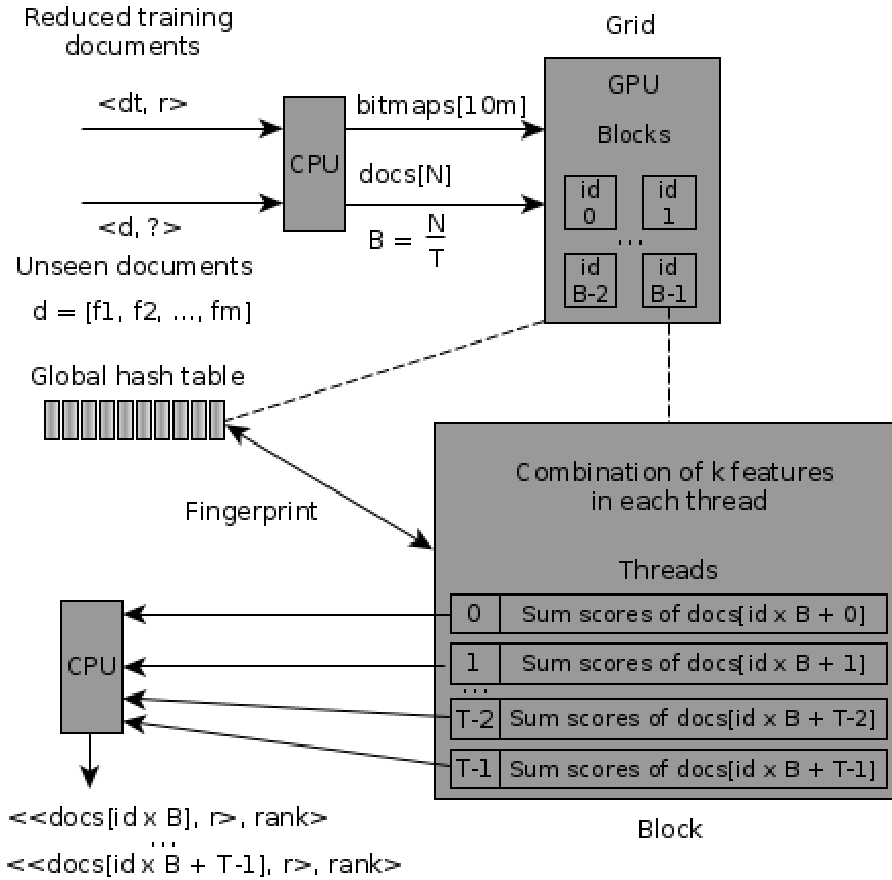

Figura 4.3: *Ilustração do PROFL sem o combinadic.*

Algorithm 18 CPU-kNNSearch(invertedIndex, q)

Require: *invertedIndex*, df, *index*, *query* $q[0..|\mathcal{V}_q|-1]$
Ensure: distance array $dist[0..|\mathbb{D}_{in}|-1]$ initialized according to the distance function used

1: array of integers $df_q[0..|\mathcal{V}_q|-1]$ initialized with zeros
2: array of integers $start_q[0..|\mathcal{V}_q|-1]$
3: **for** each term $t_i \in q$ **do**
4: $\quad df_q[i] = df[t_i]$;
5: $\quad start_q[i] = index[t_i]$;
6: **for** each term $t_j \in q$ **do**
7: $\quad end = df_q[j] + start_q[j]$;
8: $\quad$ **for** $x = start_q[j] + i$; $x < end$; $x = x + 1$ **do**
9: $\quad\quad$ uses $q[j]$ and $invertedIndex[x]$ in the partial computation of the distance between q and the document associated to $invertedIndex[x]$
10: Perform a complete sort on the distances using C++ STL Sort
11: **Return** the array *dist*.

4.5 Fine-grained Sparse and High-dimensional k-Nearest Neighbors (FiSH-kNN)

Nesta seção vamos descrever os algoritmos *manycore* do FiSH-kNN (*Fine-grained Sparse and High-dimensional k-Nearest Neighbors*), uma versão estendida do GT-kNN (GPU-based Textual k-Nearest Neighbors) desenvolvida por Canuto et. al [7].

Algorithm 19 MultiThreaded-kNN($\mathcal{E}$, Q)

Require: term-document pairs in $E[0..|\mathcal{E}|-1]$, list of queries (documents) Q
Ensure: A list of k nearest neighbors, one for each query.

1: $invertedIndex = DataIndexing(E)$;
2: **for** each query $q \in Q$ in **parallel do**
3: Read q and prepare its array;
4: $distances = CPU\text{−}kNNSearch(invertedIndex, q)$;
5: Output the first k elements of $distances$ array

Primeiro mostramos o algoritmo para a busca kNN, representando um balanceamento de carga intra-bloco, desenvolvido para checar a importância do balanceamento de carga inter-bloco (por Grid) usada no GT-kNN. Depois mostramos o algoritmo de busca do FiSH-kNN que trabalha com grupo de consultas, seu algoritmo principal e uma versão para multi-GPU.

4.5.1 Balanceamento de Carga por Bloco de Threads

O Algoritmo 20 descreve uma forma de balanço de carga considerando um bloco de *threads*. Esta forma é mais simples,e dispensa a variável $index_q$. A abordagem por bloco consiste em atribuir um termo da consulta para um bloco, onde as *threads* desse bloco trabalham em paralelo para computar as distâncias entre as entradas do índice invertido desse termo. Na linha 9, é verificado se o índice do bloco é menor que o número de termos. Isso representa um ponto fraco dependendo da natureza da consulta. Consultas com poucos termos, especialmente quando há menos termos que o número de blocos, irão ser processadas com pouco uso do hardware. Tal situação tende a ocorrer pouco no processamento de documentos.

Na linha 10 é definido o limite de iteração daquele bloco, baseado onde a entrada do termo começa no índice invertido, e quantas entradas desse termo existem pela frequência do documento. Na linha 12, cada *thread* começa em um deslocamento com seu índice, e itera até atingir o limite *end*. Outro problema surge, que é o possível desbalanceamento entre blocos, quando o termo associado à um bloco tem muito mais entradas do que o de outro bloco. Mas isso é amenizado em GPUs quando há vários blocos, pois a latência pode ser escondida nesse caso, como visto no Capítulo 2. Veremos o impacto disso na Seção de experimentos 5.4. O restante do algoritmo (linhas 9-22) é o mesmo que o Algoritmo 8.

4.5.2 Processamento de Grupo de Consultas

No Algoritmo 21 encontra-se outra otimização feita para o FiSH-kNN, o processamento de grupo de consultas. Para um tamanho de grupo $\mathcal{B}$, cada estrutura é alocada $\mathcal{B}$ vezes, e as consultas são processadas iterativamente. Como as *threads* são organizadas

Algorithm 20 Block Balanced kNNSearch(invertedIndex, q)

Require: $invertedIndex, df, index, query\ q[0..|\mathcal{V}_q|-1]$
Ensure: distance array $dist[0..|\mathbb{D}_{in}|-1]$ initialized according to the distance function used

1: array of integers $df_q[0..|\mathcal{V}_q|-1]$ initialized with zeros
2: array of integers $start_q[0..|\mathcal{V}_q|-1]$
3: **for** each term $t_i \in q$, in parallel **do**
4: $df_q[i] = df[t_i]$;
5: $start_q[i] = index[t_i]$;
6: **for** each multiprocessor $\mathcal{P}_j \in \mathcal{MP}$ **do**
7: **if** $j < |\mathcal{V}_q|$ **then**
8: $end = df_q[j] + start_q[j]$;
9: **for** each processor $p_i \in \mathcal{P}_j$ **do**
10: **for** $x = start_q[j] + i$; $x < end$; $x = x + |\mathcal{P}_j|$ **do**
11: uses $q[j]$ and $invertedIndex[x]$ in the partial computation of the distance between q and the document associated to $invertedIndex[x]$
12: Perform a partial sort on the distances using a truncated bitonic sort and a reduction operation
13: **Return** the array: $dist$ with partially sorted partitions.

em *warps*, assim que uma *warp* termina sua parte em uma consulta, ela pode partir para a próxima consulta, mesmo se houver outras *warps* processando uma consulta anterior. Então essa forma iterativa também permite um certo paralelismo entre consultas, devido a independência das *warps*. A outra vantagem consiste em enviar várias consultas de uma vez pelo barramento, que diminui a sobrecarga de transferência para a GPU. Veremos isso no algoritmo descrito na Seção 4.5.3. O restante do algoritmo FiSH-kNNSearch pode ser tanto o Algoritmo 8 quanto o Algoritmo 20, isto é, usar o balanceamento por bloco ou o por *thread* do GT-kNN.

Algorithm 21 FiSH-kNNSearch(invertedIndex, $\mathcal{G}$, $\mathcal{B}$)

Require: $invertedIndex, df, index, queries\ \mathcal{G}[0..\mathcal{B}-1][0..|\mathcal{V}_q|-1]$
Ensure: distance array $dist[0..\mathcal{B}-1][0..|\mathbb{D}_{in}|-1]$ initialized according to the distance function used

1: array of integers $df_{\mathcal{G}}[0..\mathcal{B}-1][0..|\mathcal{V}_q|-1]$ initialized with zeros
2: array of integers $index_{\mathcal{G}}[0..\mathcal{B}-1][0..|\mathcal{V}_q|-1]$
3: array of integers $start_{\mathcal{G}}[0..\mathcal{B}-1][0..|\mathcal{V}_q|-1]$
4: **for** $g \mid 0 < g < \mathcal{B}$ **do**
5: $q = \mathcal{G}[g]$;
6: $df_q = df_{\mathcal{G}}[g]$;
7: $index_q = index_{\mathcal{G}}[g]$;
8: $start_q = start_{\mathcal{G}}[g]$;
9: ...
23: Perform a partial sort on the distances using a truncated bitonic sort and a reduction operation
24: **Return** the array: $dist$ with partially sorted partitions.

A Figura 4.4 o processamento de um grupo de três consultas. As estruturas são alocadas de acordo com a maior consulta (q_1), e então é feita uma iteração sobre essas consultas.

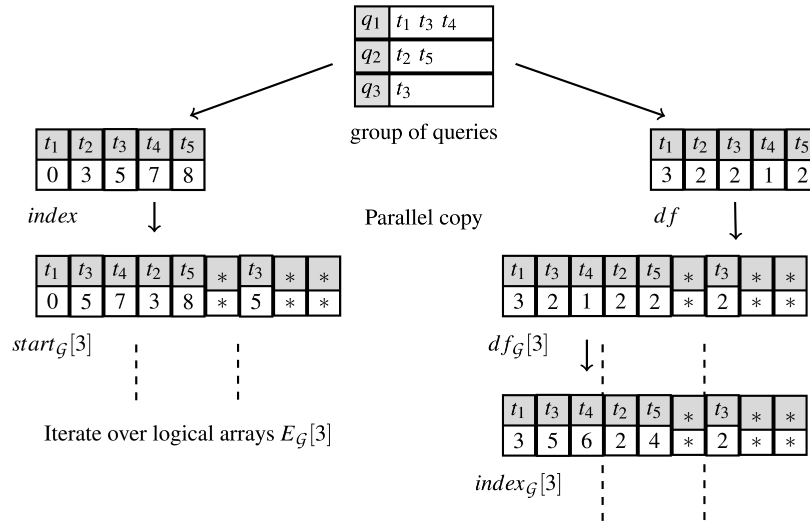

Figura 4.4: *Processamento de um grupo de três consultas.*

Algorithm 22 Fine-grained Sparse and High-dimensional-kNN($\mathcal{E}$, $\mathcal{Q}$, $\mathcal{B}$)

Require: term-document pairs in $E[0..|\mathcal{E}|-1]$, list of queries (documents) $\mathcal{Q}$, group size $\mathcal{B}$
Ensure: A list of k nearest neighbors, one for each query.

1: $invertedIndex = DataIndexing(E)$;
2: Allocate memory in GPU for the biggest query times the group size $\mathcal{B}$
3: **for** $i \mid 0 < i < \frac{|\mathcal{Q}|}{\mathcal{B}}$ **do**
4: Read $\mathcal{B}$ queries and prepare their array in $\mathcal{G}$
5: $partialLists = FiSH - kNNSearch(invertedIndex, \mathcal{G}, \mathcal{B})$;
6: **for** $j \mid 0 < j < \mathcal{B}$ **do**
7: Output k elements after sorting $partialLists[j]$ in a heap structure
8: Free the GPU memory

4.5.3 O Algoritmo FiSH-kNN

Descrevemos agora o nosso FiSH-kNN no Algoritmo 22. Sua entrada é a mesma do GT-kNN, adicionada com o tamanho do grupo de consultas escolhido. Na linha 2, durante a leitura é determinado o tamanho da maior consulta, e isso é utilizado para pré-alocar o espaço para as $\mathcal{B}$ consultas na GPU. Isso evita constantes alocações e desalocações durante a execução. Então, nas linhas 3-5, as consultas são processadas em grupos de tamanho $\mathcal{B}$, onde são enviadas para a GPU de uma vez só, e então as listas parcialmente ordenadas de cada uma são recebidas, depois de chamar o Algoritmo 21. Nas linhas 6-7 cada lista parcial é ordenada em uma *heap*, e os k vizinhos são retornados.

4.5.4 FiSH-kNN com Múltiplas GPUs

Outro aprimoramento que fizemos sobre o GT-kNN, foi o uso de múltiplas GPUs, que adaptamos nosso FiSH-kNN no Algoritmo 23. Nas linhas 1-3, cada GPU recebe os dados de entrada, e cria o mesmo índice invertido em sua memória. Cada GPU também aloca o bastante para o grupo de consultas de acordo com o tamanho da maior

consulta. Nas linhas 7-8, cada GPU recebe a sua porção de consultas a serem processadas que são processadas por grupos em cada uma, nas linhas 9-13. As paralelizações foram feitas com a API OpenMP, para associar uma *thread* de CPU para cada GPU.

Tendo descrito nossas propostas, mostraremos nossos experimentos e resultados no próximo capítulo.

Algorithm 23 Multi-GPU FiSH-kNN($\mathcal{E}$, $\mathcal{Q}$, $\mathcal{B}$, #GPUs)

Require: term-document pairs in $E[0..|\mathcal{E}|-1]$
Ensure: A list of k nearest neighbors, one for each query.

1: **for** $g \mid 0 < g < \#GPUs$, in **parallel do**
2: $set.gpu.device(g)$;
3: $invertedIndex = DataIndexing(E)$;
4: Allocate memory in each GPU for the biggest query times the group size $\mathcal{B}$
5: **for** $g \mid 0 < g < \#GPUs$, in **parallel do**
6: $set.gpu.device(g)$;
7: $N = \lceil \frac{|\mathcal{Q}|}{\#GPUs} \rceil$
8: $\mathcal{Q}^g$ = queries from $\mathcal{Q}_{g \cdot N}$ to $\mathcal{Q}_{min((g+1) \cdot N,\ |\mathcal{Q}|)}$
9: **for** $i \mid 0 < i < \frac{\lfloor \mathcal{Q}^g \rfloor}{\mathcal{B}}$ in **parallel do**
10: Read $\mathcal{B}$ queries and prepare their array in $\mathcal{G}$
11: $partialLists = FiSH - kNNSearch(invertedIndex, \mathcal{G}, \mathcal{B})$;
12: **for** $j \mid 0 < j < \mathcal{B}$ **do**
13: Output k elements after sorting $partialLists[j]$ in a heap structure
14: Free the GPUs memories

CAPÍTULO 5

Experimentos e Resultados

Neste capítulo discutimos os experimentos feitos e resultados obtidos para cada solução proposta. Na Seção 5.1 descrevemos a máquina utilizada nos experimentos. Na Seção 5.2 mostramos os resultados das implementações de aprendizado ativo PSSARP e PRSS. Na Seção 5.3 mostramos os resultados da implementação de learning to rank PROFL. E na Seção 5.4 mostramos os resultados das implementações de busca Top-K MT-kNN e FiSH-kNN.

5.1 Configuração da Máquina Utilizada

Os experimentos foram conduzidos em uma máquina rodando CentOS 7.2.1511 64-bits, com dois processadores Intel Xeon E5-2620 2GHz de 6 núcleos cada, 16GB de ECC RAM, e quatro GeForce Zotac Nvidia GTX Titan Black (*compute capability* 3.5), com 6GB de RAM e 2.880 CUDA cores (15 SMs) cada.

O código CPU foi compilado com GCC 4.8.5 , enquanto o código de GPU usou o compilador nvcc do CUDA Toolkit 7.5. Todos os códigos tiveram a *flag* de otimização O3. Durante as execuções, a máquina estava sob uso exclusivo. Os números reportados são uma média de 10 execuções.

5.2 Resultados do PSSARP e PRSS

Nesta seção apresentamos a comparação das implementações de aprendizado ativo, o SSARP e nossas propostas multicore PSSARP e manycore PRSS. No PRSS foi utilizado 256 *threads* por bloco, e $\frac{N}{256}$ blocos (Considerando N documentos). Isto permite uma concorrência de três blocos com o uso de memória compartilhada no nosso código, ao invés de um bloco com 512 *threads*. Os limiares de suporte e confiança utilizados foram 1 e 0,001, respectivamente , em todas implementações.[1]

[1]Estes são os mesmos valores usados no SSARP, LRAR e PLRAR.

Na Tabela 5.1 há a descrição dos datasets usados nos métodos de aprendizado ativo e learning to rank, mostrando o número de *queries*, *features* e de documentos em cada *fold* do treino e teste. São datasets da LETOR3[2] (TD2003, TD2004, HP2003, HP2004, NP2003 and NP2004), WEB10K[3] e Yahoo![4] Set 2, que são disponíveis livremente. Nós aplicamos o procedimento de validação cruzada com 5 *folds* [17] na LETOR3 e WEB10K, seguindo a área de aprendizado de máquina para garantir resultados confiantes. A validação cruzada divide o dataset inteiro em 5 partes, e escolhe três partes como conjunto de treino, uma como teste e a restante como conjunto de validação. O conjunto de validação foi usado pelos autores do SSARP e LRAR para encontrar os valores de suporte e confiança. Cada *fold* recebe uma atribuição diferente do outro.

	Datasets							
	TD2003	HP2003	NP2003	TD2004	HP2004	NP2004	WEB10K	Yahoo!
# Queries	10	30	30	15	15	15	2.000	3.798
# Features	59	64	64	64	64	64	131	449
Teste	9.812	29.521	29.731	14.834	14.882	14.767	240.039	103.174
Treino	29.435	88.564	89.195	44.488	44.646	44.301	720.116	34.815

Tabela 5.1: *Médias das características de cada fold dos conjuntos de dados.*

5.2.1 Características das Reduções

A Tabela 5.2 descreve a forma reduzida dos datasets. O número de partições foi escolhido de acordo com o critério de Silva et. al [33] (Ver Seção 3.2). Chamamos de reduzido-3 os datasets reduzidos por regras de tamanho até 3, e reduzido-4 para os com regras de tamanho até 4. Cada partição leva um número diferente de iterações (escolhas de documentos) até convergir, e na tabela encontra-se a média de iterações que cada partição teve, usando regras de tamanho até 3 e até 4. Como regras de tamanho 4 tem um espaço de busca maior, a convergência demora mais.

5.2.2 Tempo de Processamento das Reduções

As Tabelas 5.3, 5.4 e 5.5 descrevem os resultados de tempo dos métodos de aprendizado ativo, onde comparamos a implementação serial SSARP às versões multicore PSSARP e manycore (GPU) PRSS. Os tempos são em segundos e representam o tempo gasto para gerar o conjunto de reduzido do conjunto de treino - a quarta linha da Tabela 5.1. Para regras de tamanho 3, enquanto o SSARP pode levar centenas de segundos para

[2] https://www.microsoft.com/en-us/research/project/letor-learning-rank-information-retrieval/
[3] http://research.microsoft.com/en-us/projects/mslr/
[4] https://webscope.sandbox.yahoo.com/catalog.php?datatype=c

	Datasets reduzidos							
	TD2003	HP2003	NP2003	TD2004	HP2004	NP2004	WEB10K	Yahoo!
# Partições	5	5	5	5	5	5	13	40
Reduzido-3	671	1.091	995	593	859	658	2.101	1.514
Reduzido-4	2.082	4.936	4.545	2.115	3.499	3.262	7.337	3.039
# Iterações-3	146	222	197	133	186	173	170	50
# Iterações-4	548	1.214	1.146	515	858	827	635	106

Tabela 5.2: *Médias das características de cada $fold$ dos conjuntos de dados reduzidos.*

	Tempo total para regras de tamanho 3			Tempo total para regras de tamanho 4		
	SSARP	PRSS	Speedup	SSARP	PRSS	Speedup
TD2003	60,77	0,81	75,43	1.984,92	15,50	128,05
HP2003	340,28	2,69	126,64	25.460,00	247,05	103,06
NP2003	290,78	2,39	121,79	23.250,10	222,37	104,55
TD2004	92,81	1,12	82,70	3.308,07	28,24	117,15
HP2004	152,71	1,49	102,66	8.558,08	82,89	103,24
NP2004	135,20	1,38	97,97	7.873,68	75,94	103,69
WEB10K	3.602,44	51,01	70,63	56.111,10	1.398,34	40,13
Yahoo!	227,09	3,76	60,40	2.120,33	17,21	123,20

Tabela 5.3: *Tempo de execução em segundos das implementações SSARP e PRSS.*

terminar, o PRSS leva apenas poucos segundos nos datasets da LETOR. O tempo da WEB10K abaixa de 1 hora (CPU serial) para menos 1 minuto (GPU). Como as regras de tamanho 3 tem menos combinações, o PRSS termina bem rápido nos datasets menores. Apesar do Yahoo! ter mais *features*, seus documentos são diversos o bastante para fazer com que a convergência seja atingida bem rápido. Os resultados do PRSS são muito bons, chegando a ter até 128x de *speedup* em relação ao SSARP. O *speedup* com regras de tamanho 4 chega a diminuir em alguns casos, devido a divergência de *threads* ocorrer com mais frequência. Esse impacto é maior na WEB10K, já que tem um número alto de *features* e levou mais tempo para convergir, tendo seu *speedup* reduzido para 40x.

Na Tabela 5.4 descrevemos os resultados do PRSS com uma implementação multi-GPU, para 2 (PRSS-2) e 4 (PRSS-4) GPUs. Os resultados são para regras de tamanho 4 para LETOR e Yahoo!, já que não houve ganhos para tamanho 3, por ter menos interseções a serem feitas. Somente a WEB10K teve ganhos com regras de tamanho 3 devido ao seu tamanho, aumentando o *speedup* para 136x sobre o SSARP. A implementação multi-GPU realmente ajuda no processamento do PRSS ao usar regras de tamanho 4, aumentando o *speedup* dos datasets da LETOR para poucas centenas sobre o SSARP, atingindo até 431x.

Na Tabela 5.5 encontra-se os resultados do PSSARP, executado com 12 *threads*. Com regras de tamanho 3, também é possível ver que para esse tamanho as iterações são bem rápidas, fazendo o trabalho paralelo ficar pequeno perto do serial. Isso acaba por fazer o *speedup* ficar apenas em torno de 6, deixando a execução na casa de dezenas de

	Tempo total		Speedup-2	Speedup-4
	PRSS-2	PRSS-4	SSARP	SSARP
TD2003	7,85	6,23	252,97	318,58
HP2003	133,29	79,75	191,02	319,25
NP2003	120,26	72,26	193,33	321,74
TD2004	16,69	7,89	198,20	419,44
HP2004	49,26	19,85	173,73	431,16
NP2004	45,19	18,39	174,24	428,06
WEB10K-3	33,03	26,33	109,06	136,84
WEB10K-4	711,78	372,52	78,83	150,62
Yahoo!	10,40	9,40	203,88	225,56

Tabela 5.4: *Speedup do PRSS com múltiplas GPUs usando regras de tamanho 4.*

segundos para a LETOR, e acima de 600 para a WEB10K. Ao usar regras de tamanho 4, o trabalho paralelo aumenta, e para todas coleções o *speedup* fica quase em 11x, próximo do ideal. Mas o PRSS mostra a superioridade do processamento em GPU ao gastar até 17x menos tempo que o PSSARP com 12 *threads* (Em relação a HP2003) .

	Tempo total para regras de tamanho 3			Tempo total para regras de tamanho 4		
	SSARP	PSSARP	Speedup	SSARP	PSSARP	Speedup
TD2003	60,77	9,55	6,36	1.984,92	183,66	10,81
HP2003	340,28	46,60	7,30	25.460,00	2.376,37	10,71
NP2003	290,78	40,69	7,15	23.250,10	2.157,26	10,78
TD2004	92,81	13,95	6,65	3.308,07	305,52	10,83
HP2004	152,71	20,51	7,44	8.558,08	787,84	10,86
NP2004	135,20	18,53	7,30	7.873,68	728,24	10,81
WEB10K	3.602,44	635,00	5,67	56.111,10	5.488,54	10,22
Yahoo!	227,09	35,61	6,37	2.120,33	203,67	10,41

Tabela 5.5: *Tempo de execução em segundos das implementações SSARP e PSSARP.*

5.3 Resultados do PROFL

Nesta seção mostramos os resultados da tarefa de ranqueamento, usando ambas versões reduzidas das coleções da Tabela 5.2, reduzido-3 e reduzido-4, e aplicando o método de *learning to rank* com regras de tamanho 3 e 4 em cada uma. Nas Tabelas 5.6 e 5.7, confirmamos que nossa proposta PROFL supera os outros métodos. Mesmo usando apenas uma GPU, o *speedup* sobre o método sequencial LRAR chega a 508x, e 9,2x sobre o paralelo (GPU) PLRAR. O tempo para criar os *bitmaps* do conjunto de treino não está incluído nestas tabelas, que são de 0,15, 0,4 e 2 segundos para a LETOR3, Yahoo! e WEB10K, respectivamente. Graças a nossa proposta PRSS, essa é a primeira vez que um método de ranqueamento baseado em regras de associação foi aplicado nos datasets WEB10K e Yahoo!.

5.3.1 Tempo de Processamento do PROFL

A Tabela 5.6 mostra os resultados do PROFL usando uma GPU. Enquanto o PROFL processa uma consulta em 0,2 segundo, o LRAR leva 500x mais tempo no melhor caso, que seria 100 segundos (Ver Tabela 5.9). Este resultado mostra como nossa proposta é apropriada para fazer processamento sob-demanda. Usando os conjuntos reduzido-4, o tempo do PROFL aumenta menos do que o PLRAR nos conjuntos da LETOR3, e ambos aumentam igualmente no Yahoo!, como visto na parte inferior da Tabela 5.6. O *speedup* sobre o LRAR reduz na maioria dos casos já que o aumento do número de documentos afeta o tamanho do *bitmap*, e consequentemente, o espaço disponível na cache do PROFL. Apesar do PROFL ter se sobressaído contra os outros algoritmos, há pouca diferença em relação ao PLRAR quando são usados apenas regras de tamanho 3 nos conjuntos LETOR3. Isso porque há poucas combinações a serem feitas, reduzindo a quantidade de trabalho paralelo. Com os conjuntos WEB10K e Yahoo! e usando regras de tamanho 4, há uma performance melhor do PROFL. Isso mostra as vantagens do uso da abordagem de bloco de *threads*, da cache paralela, e dos *bitmaps* vetorizados.

Há um número maior de *features* e documentos nos conjuntos WEB10K e Yahoo!. Assim, o LRAR sofre mais com o aumento de *features*, demorando mais para processar um documento, o que faz o PROFL set até 441x mais rápido. Ao processar regras de tamanho 4, o *speedup* sobre o PLRAR é de 9,2x no melhor caso, mostrando que o PROFL é uma implementação superior em GPU. Quando o LRAR foi executado com regras de tamanho 4, apenas 10% do primeiro *fold* foi processado depois de 20 horas, mostrando que é incapaz de processar os conjuntos WEB10K e Yahoo!.

As Tabelas 5.7 e 5.8 mostram os resultados do PROFL com multi-GPU. Nessa implementação, somente regras de tamanho 4 que tiveram ganho de *speedup*. O *speedup* do PROFL na LETOR contra o PLRAR foi até de 19,3x e 27,3x para 2 e 4 GPUs respectivamente, quando usando os conjuntos reduzido-4. Contra o LRAR o *speedup* sobe alguma centenas. Isso mostra que a escalabilidade para multiGPU é maior com a abordagem por bloco de *threads*, e quando a consulta tem poucos documentos. Considerando a WEB10K e Yahoo!, o PROFL teve um *speedup* quase linear para 2 e 4 GPUs, devido ao aumento de cache disponível com mais GPUs. Os resultados para os conjuntos reduzido-3 foram similares, e o uso regras de tamanho 3 não teve ganhos devido ao gargalo do tempo de entrada e envio de dados. Esses gargalos tiveram um impacto maior nos conjuntos LETOR, onde os speedpus não passaram de 2,7x.

5.3.2 Resultados do PROFL com abordagem por *threads*

Discutiremos os resultados do PROFL sem o uso do combinadic e seguindo uma abordagem por *threads* como no PLRAR, nas Tabelas 5.9 e 5.10. Chamaremos essa

	Usando os conjuntos reduzido-3							
	Regras de tamanho 3				Regras de tamanho 4			
	Tempo para 1 query		Speedup		Tempo para 1 query		Speedup	
	PROFL	PLRAR	PLRAR	LRAR	PROFL	PLRAR	PLRAR	LRAR
TD2003	**0,09**	0,13	1,39	26,11	**0,18**	0,61	3,35	445,28
HP2003	**0,08**	0,10	1,32	31,52	**0,16**	0,61	3,81	507,96
NP2003	**0,08**	0,10	1,31	31,26	**0,16**	0,60	3,71	498,09
TD2004	**0,09**	0,12	1,31	30,28	**0,21**	0,60	2,93	496,92
HP2004	**0,10**	0,13	1,32	29,38	**0,21**	0,67	3,20	506,31
NP2004	**0,09**	0,12	1,31	28,83	**0,21**	0,64	3,09	491,42
WEB10K	**0,02**	0,11	5,73	119,66	**0,64**	5,89	9,27	*
Yahoo!	**0,02**	0,13	5,87	441,39	**5,21**	25,64	4,92	*
	Usando os conjuntos reduzido-4							
	PROFL	PLRAR	PLRAR	LRAR	PROFL	PLRAR	PLRAR	LRAR
TD2003	**0,10**	0,18	1,82	24,71	**0,21**	2,42	11,79	461,39
HP2003	**0,08**	0,17	1,97	28,09	**0,29**	2,98	10,19	293,04
NP2003	**0,08**	0,16	1,92	28,93	**0,28**	2,93	10,54	297,89
TD2004	**0,10**	0,17	1,67	28,48	**0,25**	2,59	10,28	436,88
HP2004	**0,10**	0,19	1,81	27,11	**0,29**	3,13	10,89	372,84
NP2004	**0,10**	0,18	1,79	27,26	**0,26**	2,98	11,35	397,27
WEB10K	**0,02**	0,15	7,72	134,52	**1,95**	7,94	4,07	*
Yahoo!	**0,02**	0,18	7,15	403,92	**8,35**	40,83	4,89	*

Tabela 5.6: *Tempo de execução em segundos do PROFL. Melhores resultados estão em negrito.*

	Usando os conjuntos reduzido-3					
	Tempo para 1 query		Speedup-2		Speedup-4	
	PROFL-2	PROFL-4	PLRAR	LRAR	PLRAR	LRAR
TD2003	0,14	0,12	4,32	574,31	5,20	691,68
HP2003	0,12	0,10	5,20	692,27	6,37	849,18
NP2003	0,12	0,10	5,07	680,89	6,20	832,40
TD2004	0,15	0,12	4,02	681,76	4,97	842,28
HP2004	0,15	0,12	4,41	696,54	5,44	860,19
NP2004	0,15	0,12	4,25	674,76	5,23	831,63
WEB10K	0,33	0,17	18,02	*	34,06	*
Yahoo!	2,65	1,37	9,69	*	18,78	*
	Usando os conjuntos reduzido-4					
	Tempo para 1 query		Speedup-2		Speedup-4	
	PROFL-2	PROFL-4	PLRAR	LRAR	PLRAR	LRAR
TD2003	0,16	0,13	15,09	590,66	18,55	726,32
HP2003	0,16	0,11	18,30	526,30	27,32	785,70
NP2003	0,15	0,11	19,30	545,72	26,99	763,19
TD2004	0,16	0,13	15,99	679,69	20,67	878,63
HP2004	0,17	0,13	17,98	615,48	23,94	819,37
NP2004	0,17	0,13	17,62	616,76	23,69	829,56
WEB10K	0,99	0,50	8,04	*	15,77	*
Yahoo!	4,23	2,13	9,66	*	19,16	*

Tabela 5.7: *Speedup do PROFL com múltiplas GPUs sobre o LRAR e PLRAR, usando regras de tamanho 4.*

versão de PROFL_T (PROFL_$Thread{-}approach$). Em seguida mostramos o *speedup* do PROFL sobre o PROFL_T na Tabela 5.11.

Na Tabela 5.9 mostramos o tempo de execução do PROFL_T e LRAR, e os

	Conjuntos reduzido-3	
	Speedup	
	PROFL-2	PROFL-4
TD2003	1,30	1,55
HP2003	1,36	1,66
NP2003	1,36	1,67
TD2004	1,37	1,70
HP2004	1,37	1,69
NP2004	1,37	1,70
WEB10K	1,94	3,68
Yahoo!	1,97	3,82
	Conjuntos reduzido-4	
	PROFL-2	PROFL-4
TD2003	1,28	1,57
HP2003	1,80	2,68
NP2003	1,83	2,56
TD2004	1,56	2,01
HP2004	1,65	2,20
NP2004	1,55	2,09
WEB10K	1,98	3,87
Yahoo!	1,98	3,92

Tabela 5.8: *Speedup do PROFL com múltiplas GPUs sobre uma GPU, usando regras de tamanho 4.*

speedups do PROFL_T sobre PLRAR e LRAR. Nas coleções LETOR com regras de tamanho 3 os *speedups* são pequenos sobre o PLRAR, e algumas dezenas sobre o LRAR. Com a WEB10K e Yahoo! a cache consegue fazer o *speedup* ficar acima de 5x sobre o PLRAR,e 100x sobre o LRAR. Com regras de tamanho 4 os conjuntos LETOR obtiveram *speedups* entre 3x e 7x, para os conjuntos reduzido-3 e reduzido-4 respectivamente, sobre o PLRAR, mostrando a efetividade da cache nessas coleções. Mas a implementação por abordagem de *threads* sofre mais com o problema de divergência de *threads* quanto maior o tamanho da regra e quantidade de *features*, como podemos ver pelo baixo *speedup* de 1,1x na Yahoo! em ambas versões reduzidas. A versão reduzida-4 da WEB10K tem mais que 3x o número de documentos de treino, e isso reduz bastante o número de posições disponíveis na cache.

Com o uso de múltiplas GPUs, também mostramos apenas os resultados com regras de tamanho 4 na Tabela 5.10, já que o *speedup* na LETOR diminui e na WEB10K e Yahoo! aumentam em apenas 10%. Com 2 GPUs o *speedup* na LETOR aumenta para até 4,5x, mas devido ao menor potencial de escalabilidade da abordagem por *threads*, com 4 GPUs apenas as maiores coleções HP2003 e NP2003 ficam mais rápidas, enquanto o restante tem um *speedup* menor do que com 2 GPUs. Com uma coleção maior como a WEB10K o *speedup* fica ideal, e com 4 GPUs a Yahoo! ganha um *speedup* superlinear por ter mais espaço de cache disponível. Esse efeito ocorre na WEB10K com o conjunto reduzido-4, mas pelo *bitmap* ficar 3 vezes maior com o aumento do conjunto.

Na Tabela 5.11 mostramos o *speedup* do PROFL sobre o PROFL_T. Nos

	Usando os conjuntos reduzido-3							
	Regras de tamanho 3				Regras de tamanho 4			
	Tempo para 1 query		Speedup		Tempo para 1 query		Speedup	
	PROFL_T	LRAR	PLRAR	LRAR	PROFL_T	LRAR	PLRAR	LRAR
TD2003	0,13	2,43	1,00	18,78	0,22	81,66	2,82	375,87
HP2003	0,09	2,38	1,15	27,35	0,16	81,48	3,94	525,37
NP2003	0,09	2,36	1,14	27,07	0,16	80,36	3,82	512,91
TD2004	0,12	2,83	1,05	24,20	0,21	102,48	2,86	484,79
HP2004	0,12	2,81	1,06	23,55	0,22	106,27	3,10	490,22
NP2004	0,12	2,70	1,05	23,05	0,21	101,19	3,04	482,86
WEB10K	0,02	2,25	5,60	116,89	0,95	*	6,21	*
Yahoo!	0,02	9,86	5,93	446,13	23,19	*	1,11	*
	Usando os conjuntos reduzido-4							
	PROFL_T	LRAR	PLRAR	LRAR	PROFL_T	LRAR	PLRAR	LRAR
TD2003	0,16	2,45	1,10	14,89	0,38	94,87	6,37	249,31
HP2003	0,11	2,38	1,50	21,43	0,45	85,61	6,62	190,26
NP2003	0,11	2,44	1,48	22,20	0,44	82,75	6,69	189,29
TD2004	0,15	2,90	1,14	19,46	0,36	110,07	7,18	305,25
HP2004	0,15	2,82	1,24	18,62	0,42	107,18	7,37	252,30
NP2004	0,15	2,78	1,23	18,65	0,40	104,19	7,43	260,26
WEB10K	0,02	2,60	7,50	130,69	3,77	*	2,10	*
Yahoo!	0,03	10,01	6,52	368,30	34,20	*	1,19	*

Tabela 5.9: *Tempo de execução em segundos do PROFL com abordagem por threads.*

conjuntos reduzido-3, com 1 GPU o desempenho de ambos é próximo, e com regras de tamanho 4 o *speedup* chega a ser 4,45x para a Yahoo!, mostrando que para um alto número de *features*, a divergência de *threads* tem um impacto menor na abordagem por blocos. Em quase todos os casos, o uso de múltiplas GPUs teve *speedups* em torno de 2x por essa abordagem também permitir uma maior ocupação das GPUs.

5.3.3 Análise do Uso da Cache

Nesta subseção discutiremos sobre a cache usando as Tabelas 5.12 e 5.13. Os valores estão em notação científica. A Tabela 5.12 mostra a quantidade de posições disponíveis na cache utilizando 4,5GB de RAM, ou seja, quantos *bitmaps* podem ser armazenados. Esse valor de RAM foi o maior possível, considerando que estruturas auxiliares utilizaram o restante. Para as coleções LETOR foi utilizado o *bitmap* da HP2003, o que dá 26 milhões de posições para o conjunto reduzido-3. Para a LETOR e WEB10K a versão reduzida-4 tem três vezes menos posições, e para a Yahoo! duas vezes. O impacto disso no tempo de processamento pôde ser visto no início desta Seção 5.3.

Levando em consideração os acertos (hit) e erros (miss) nos acessos à cache, a

	Usando os conjuntos reduzido-3					
	Tempo para 1 query		Speedup-2		Speedup-4	
	PROFL_T-2	PROF_TL-4	PLRAR	LRAR	PLRAR	LRAR
TD2003	0,21	0,24	2,91	387,84	2,58	343,10
HP2003	0,13	0,13	4,55	606,68	4,74	631,71
NP2003	0,13	0,13	4,44	596,13	4,68	628,58
TD2004	0,18	0,19	3,44	583,60	3,19	540,19
HP2004	0,18	0,20	3,67	579,53	3,38	533,75
NP2004	0,18	0,19	3,50	556,16	3,27	519,25
Web10K	0,46	0,23	12,83	*	25,62	*
Yahoo!	11,16	4,53	2,30	*	5,67	*
	Usando os conjuntos reduzido-4					
	Tempo para 1 query		Speedup-2		Speedup-4	
	PROFL_T-2	PROFL_T-4	PLRAR	LRAR	PLRAR	LRAR
TD2003	0,30	0,31	8,01	313,49	7,77	304,20
HP2003	0,27	0,20	11,18	321,48	14,82	426,30
NP2003	0,26	0,20	11,06	312,79	14,85	419,77
TD2004	0,27	0,25	9,53	404,93	10,21	434,13
HP2004	0,30	0,29	10,27	351,55	10,97	375,64
NP2004	0,30	0,28	9,95	348,53	10,50	367,52
WEB10K	1,62	0,76	4,91	*	10,44	*
Yahoo!	18,05	8,24	2,26	*	4,95	*

Tabela 5.10: *Speedup do PROFL_T com múltiplas GPUs usando regras de tamanho 4.*

Tabela 5.13 mostra esses dois valores para as regras de tamanho 3 e 4. De tamanho 2 já são pré-computadas. Os valores reportados são do PROFL, enquanto o PROFL_T teve valores similares variando em até +-5%. Em versões reduzidas, usando regras de tamanho 3, a quantidade de hit foi aproximadamente duas ordens de magnitude maior que a de miss nas coleções LETOR, e 4 ordens de magnitude na WEB10K e Yahoo!. Esses dois maiores conjuntos realmente foram os que tiveram maior *speedup* com regras de tamanho 3 na Tabela 5.6.

Com regras de tamanho até 4, o hit e miss das regras de tamanho 3 nas coleções da LETOR mantêm na mesma diferença. O hit aumenta em até 33% no conjunto reduzido-3 e em até 80% no reduzido-4, que são os casos onde o prefixo do *itemset* é buscado na cache com as regras de tamanho 4, o hit também é quase duas ordens de magnitude maior que o miss, nas duas versões reduzidas. Com o número menor de posições no conjunto reduzido-4, o miss ficou até 3x maior do que no reduzido-3.

Para a WEB10K, com regras de tamanho até 4, a versão reduzida-3 teve 2,5x mais hit do que miss nas regras de tamanho 4, e 1,3x mais hits de regras de tamanho 3, mostrando que os prefixos foram bem mais aproveitados. A quantidade de diferentes *itemsets* e do tamanho do *bitmap* fez a proporção hit/miss ser menor do que da LETOR. Isso é ainda mais visível no conjunto Yahoo!, onde o miss de regra de tamanho 4

	Usando os conjuntos reduzido-3							
	Speedup com regras de tamanho 3							
	TD2003	HP2003	NP2003	TD2004	HP2004	NP2004	WEB10K	Yahoo!
PROFL_T	1,39	1,15	1,15	1,25	1,25	1,25	1,02	0,99
PROFL_T-2	1,59	1,23	1,25	1,40	1,37	1,40	1,02	1,01
PROFL_T-4	2,03	1,41	1,42	1,67	1,64	1,68	1,02	1,04
	Speedup com regras de tamanho 4							
PROFL_T	1,18	1,00	1,00	1,03	1,03	1,02	1,49	4,45
PROFL_T-2	1,49	1,14	1,14	1,17	1,20	1,21	1,40	4,22
PROFL_T-4	2,01	1,34	1,33	1,56	1,61	1,60	1,33	3,32
	Usando os conjuntos reduzido-4							
	Speedup com regras de tamanho 3							
PROFL_T	1,66	1,31	1,30	1,46	1,46	1,46	1,03	1,10
PROFL_T-2	2,01	1,49	1,50	1,75	1,69	1,75	1,04	1,11
PROFL_T-4	2,78	1,87	1,86	2,28	2,22	2,29	1,06	1,13
	Speedup com regras de tamanho 4							
PROFL_T	1,85	1,54	1,57	1,43	1,48	1,53	1,93	4,09
PROFL_T-2	1,88	1,64	1,74	1,68	1,75	1,77	1,64	4,27
PROFL_T-4	2,39	1,84	1,82	2,02	2,18	2,26	1,51	3,87

Tabela 5.11: *Speedup do PROFL sobre o PROFL com abordagem por threads.*

	# Posições	
	Reduzido-3	Reduzido-4
LETOR3	26.810.182	7.021.715
WEB10K	14.043.429	4.336.942
Yahoo!	19.660.801	10.922.667

Tabela 5.12: *Quantidade de posições disponíveis na cache.*

é maior que o hit, mas o de tamanho 3 é 4 ordens de magnitude menor que o seu respectivo hit, mostrando que o conjunto do Yahoo! teve a maior parte do uso da cache no aproveitamento de prefixos. A versão reduzida-4 desses dois conjuntos segue o mesmo comportamento, devido ao aumento do *bitmap*.

Com o uso de múltiplas GPUs, apenas o conjunto Yahoo! teve diferenças significativas e com regras de tamanho 4. O miss 3 e hit 3 tiveram uma redução de 7% e 14%, com 2 e 4 GPUs, respectivamente, em ambas versões reduzidas. Em relação ao hit 4, no conjunto reduzido-3 o aumento do hit foi de 20% e 44%, com 2 e 4 GPUs, respectivamente. O miss 4 foi reduzido em 7% e 15% com 2 e 4 GPUs. No conjunto reduzido-4 os ganhos de hit4 foram de 27% e 60% com 2 e 4 GPUs, e uma redução de 4% e 10% do miss 4 com e 4 GPUs, respectivamente. Estes ganhos são devido ao fato de haver mais memória para a cache em uma configuração multi-GPU.

	TD2003	HP2003	NP2003	TD2004	HP2004	NP2004	WEB10K	Yahoo
	Usando os conjuntos reduzido-3							
	Hit e miss com regras de tamanho 3							
hit 3	1,66E+07	5,03E+07	5,07E+07	2,97E+07	2,98E+07	2,95E+07	2,04E+09	1,04E+10
miss 3	1,61E+05	2,04E+05	1,98E+05	2,35E+05	2,31E+05	2,24E+05	9,13E+05	7,56E+06
	Hit e miss com regras de tamanho 4							
hit 3	2,43E+07	6,29E+07	6,32E+07	3,84E+07	3,95E+07	3,87E+07	2,65E+10	1,16E+12
miss 3	1,58E+05	2,01E+05	1,94E+05	2,26E+05	2,28E+05	2,15E+05	9,09E+05	1,63E+08
hit 4	3,11E+08	9,47E+08	9,54E+08	6,09E+08	6,10E+08	6,06E+08	6,35E+10	3,99E+11
miss 4	7,71E+06	1,26E+07	1,25E+07	8,72E+06	9,69E+06	9,18E+06	2,44E+10	1,15E+12
	Usando os conjuntos reduzido-4							
	Hit e miss com regras de tamanho 3							
hit 3	1,66E+07	5,03E+07	5,06E+07	2,96E+07	2,97E+07	2,95E+07	2,04E+09	1,04E+10
miss 3	1,81E+05	2,47E+05	2,30E+05	2,73E+05	2,79E+05	2,88E+05	9,27E+05	8,79E+06
	Hit e miss com regras de tamanho 4							
hit 3	2,60E+07	9,07E+07	8,98E+07	4,60E+07	4,75E+07	4,55E+07	4,89E+10	1,34E+12
miss 3	1,65E+05	2,29E+05	2,13E+05	2,36E+05	2,46E+05	2,52E+05	1,06E+06	1,75E+09
hit 4	3,10E+08	9,19E+08	9,27E+08	6,01E+08	6,02E+08	5,99E+08	4,10E+10	2,10E+11
miss 4	9,41E+06	4,04E+07	3,91E+07	1,64E+07	1,78E+07	1,60E+07	4,69E+10	1,34E+12

Tabela 5.13: *Cache hit e miss do PROFL.*

5.4 Resultados do MT-kNN e FiSH-kNN

Nesta seção descrevemos os resultados obtidos com as implementações de kNN: o MT-kNN com variado número de *threads* e o FiSH-kNN com variado números de GPUs, tamanhos de grupo de consulta, e comparação das estratégias de balanceamento de carga. Para comparação utilizamos o G-kNN (GPU-based kNN) [30], uma implementação em GPU do kNN para bases esparsas e de alta dimensionalidade, que também faz uma indexação dos dados e utiliza a biblioteca Thrust para fazer ordenação. Porém, sua implementação consiste no cálculo de distâncias para todos documentos, diferentemente do índice invertido onde o cálculo é feito somente com as entradas que possuem o termo em comum. Uma outra implementação sequencial que foi escolhida para comparação foi a ferramenta Rainbow[5], utilizando seu classificador kNN também baseado em índice invertido. Para uma comparação mais justa, alteramos o código fonte para remover a parte de votação (classificação) do Rainbow, e retornar apenas os k vizinhos mais próximos.

Em todas implementações o valor de K utilizado foi 30, e a função de distância foi a do cosseno. Em nossa implementação do FiSH-kNN e da indexação de dados, utilizamos a biblioteca Thrust para fazer as somas de prefixo paralelas. Veremos o impacto desta escolha nos próximos parágrafos. A execução do FiSH-kNN e GT-kNN usaram o

[5]Disponível em https://people.cs.umass.edu/~mccallum/bow/rainbow/.

mesmo número de blocos (3 vezes o número de SMs) e de *threads* por bloco (512). O número de blocos influencia na quantidade de partições da ordenação parcial, e aumento o consumo de memória.

Para avaliar a busca kNN, foram utilizados os seguintes seis datasets textuais reais: 20 Newsgroups (20NG), Four Universities (4Uni), Reuters90 (Reut90), ACM Digital Library (ACM), Medline (Med) e RCV1. Estes datasets foram pré-processados da seguinte forma: foram removidas as *stopwords* usando a lista padronizada SMART[6], e foram removidos todos termos com frequência de documento (*Document Frequency*) menor que 6. A ponderação dos termos usada foi TF-IDF. Este pré-processamento é o mesmo feito por Canuto et al. [7].

Na Tabela 5.14 temos os detalhes dos 6 datasets. Os maiores são Medline e RCV1, mas todos tem a característica de serem bem esparsos e de alta dimensionalidade. A densidade representa o número médio de termos que cada documento tem naquele dataset. A coluna de número de consultas mostra a quantidade de amostras retiradas sem reposição da coleção, que serão usadas para testar o tempo de consulta dos métodos kNN. Cada execução foi uma amostragem aleatória diferente de consultas, onde todos documentos restantes são indexados. Os números escolhidos levam em consideração o número de documentos que o dataset tem, e o tempo de processamento que alguns métodos levariam para executar (Como 40 horas na RCV1 completa usando o G-kNN).

Dataset	# Termos	# Documentos	Densidade	Tamanho	# Consultas
20NG	61.032	18.766	129,74	30MB	3.500
4Uni	40.192	8.274	139,81	14MB	1.500
ACM	56.413	24.897	27,76	8.5MB	5.000
Med	268.766	861.454	30,88	327MB	10.000
RCV1	134.914	804.427	78,40	884MB	10.000
Reuters90	19.572	13.327	77,26	13MB	2.500

Tabela 5.14: *Informação geral dos datasets textuais.*

5.4.1 Tempo de Processamento de uma Consulta

Na Tabela 5.15 temos o tempo médio de uma consulta, dado em milissegundos. Executamos o MT-kNN de forma sequencial (compilado sem as flags do OpenMP) e com 6 e 12 *threads*, e o FiSH-kNN com 1, 2 e 4 GPUs, como mostram os números depois do hífen na primeira coluna. Nesta execução o FiSH-kNN foi executado com o mesmo balanceamento de carga do GT-kNN (balanceamento inter-bloco) e o tamanho de grupo de consulta foi 1. Podemos ver que o tempo do FiSH-kNN é um pouco menor que do GT-kNN nesse caso, que é o tempo salvo ao alocar as estruturas apenas uma vez para todas consultas. São tempos bastante pequenos mesmo para as bases maiores RCV1 e

[6]Disponível em `http://www.lextek.com/manuals/onix/stopwords2.html`.

Med. É visível o grande impacto da comparação com todas entradas no G-kNN, mesmo sendo em GPU, perde para o MT-kNN sequencial. Mesmo contra o Rainbow, o MT-kNN sequencial gasta bem menos tempo em alguns datasets, mostrando que o índice invertido utilizado e a busca kNN é superior ao dessa ferramenta. Ao usar 12 *threads*, o MT-kNN teve uma leve perda na RCV1, o que indica que houve uma saturação de acessos ao índice. As soluções multi-GPU apresentaram um tempos perto dos ideias, em torno de 2x menor para 2 GPUS, e 4x para 4 GPUs.

	20NG	4Uni	ACM	Medline	RCV1	Reuters90
Rainbow	38,43	15,34	1,57	48,98	455,45	13,93
MT-kNN - 1	4,55	1,81	0,88	42,05	118,80	2,24
MT-kNN - 6	0,81	0,35	0,16	8,51	26,07	0,42
MT-kNN - 12	0,46	0,22	0,10	7,61	28,26	0,27
G-KNN	6,95	3,75	1,97	49,59	205,98	2,56
GT-kNN	1,31	1,22	0,95	3,67	7,53	1,06
FiSH-kNN - 1	0,55	0,50	0,40	3,42	6,95	0,43
FiSH-kNN - 2	0,29	0,28	0,20	1,69	3,38	0,22
FiSH-kNN - 4	0,17	0,16	0,11	0,85	1,69	0,13

Tabela 5.15: *Tempo em milissegundos para uma consulta.*

Na Tabela 5.16 temos os tempos de indexação de cada método, em segundos. Para as bases pequenas o MT-kNN não tem melhora, mas para as grandes o ganho é visível, caindo de 4,58s para 1,35s. O G-kNN não possui índice invertido, então sua indexação é praticamente apenas a cópia dos dados para a GPU. Para o Rainbow, no seu tempo está incluído o tempo de leitura, pois não conseguimos separar do tempo de indexação. Podemos ver o impacto do uso da biblioteca Thrust ao invés da CUDPP, onde os tempos deveriam ser iguais para o FiSH-kNN-1 e GT-kNN, um *overhead* médio de 0,22s é acrescentado nos tempos do FiSH-kNN. Ao utilizar múltiplas GPUs, a saturação do barramento faz o tempo aumentar, devido a cópia das entradas para todas GPUs.

	20NG	4Uni	ACM	Medline	RCV1	Reuters90
Rainbow	1,93	1,05	0,68	23,12	53,66	0,87
MT-kNN - 1	0,12	0,06	0,03	1,89	4,58	0,05
MT-kNN - 6	0,06	0,03	0,02	0,82	2,07	0,03
MT-kNN - 12	0,04	0,02	0,01	0,58	1,35	0,02
G-KNN	0,35	0,35	0,37	0,49	0,64	0,35
GT-kNN	0,02	0,01	0,01	0,30	0,70	0,01
FiSH-kNN - 1	0,24	0,23	0,22	0,55	1,04	0,22
FiSH-kNN - 2	0,52	0,52	0,49	0,84	1,27	0,48
FiSH-kNN - 4	0,98	1,00	0,96	1,38	2,08	0,95

Tabela 5.16: *Tempo em segundos da indexação.*

Na Tabela 5.17 está o *speedup* do FiSH-kNN com 1 GPU em relação aos outros métodos, de acordo com os tempos da Tabela 5.15. Para as bases menores, o MT-kNN com 6 e 12 *threads* chega a ser mais rápido que o FiSH-kNN, devido as sobrecargas de usar GPU. Em relação ao Rainbow o *speedup* chegou até 69x, e até 30x sobre o G-kNN, mostrando a eficácia de se usar um índice invertido. Em relação ao GT-kNN os *speedups*

são de até 2,4x devido a única alocação que é feita, mas o impacto é menor nas bases maiores.

	20NG	4Uni	ACM	Medline	RCV1	Reuters90
Rainbow	69,53	30,70	3,97	14,30	65,53	32,12
MT-kNN - 1	8,23	3,62	2,23	12,28	17,09	5,15
MT-kNN - 6	1,46	0,71	0,40	2,49	3,75	0,97
MT-kNN - 12	0,83	0,43	0,26	2,22	4,07	0,62
G-KNN	12,58	7,50	4,97	14,48	29,64	5,89
GT-kNN	2,36	2,45	2,39	1,07	1,08	2,45

Tabela 5.17: *Speedup do FiSH-kNN em relação aos outros.*

Em relação ao FiSH-kNN multi-GPU a Tabela 5.18 mostra que os *speedup* são ideais para a Med e RCV1. Isso mostra que a simples solução de dividir as consultas entre as GPUs é suficiente, e também reforça a independência do processamento de cada documento.

	20NG	4Uni	ACM	Medline	RCV1	Reuters90
FiSH-kNN - 2	1,90	1,76	1,93	2,03	2,06	1,96
FiSH-kNN - 4	3,33	3,06	3,53	4,04	4,10	3,39

Tabela 5.18: *Speedup do FiSH-kNN ao usar múltiplas GPUs.*

Comparando as soluções em CPU, na Tabela 5.19 mostra o *speedup* do MT-kNN sequencial em relação ao Rainbow, e o MT-kNN paralelo em relação ao sequencial. O *speedup* sobre o Rainbow chegou até 8x, mas nas bases de menor densidade ACM e Medline os *speedups* não passaram de 2x. Isso quer dizer que consultas maiores favorecem mais o MT-kNN e o uso do seu índice invertido.

	20NG	4Uni	ACM	Medline	RCV1	Reuters90
MT-kNN1/Rainbow	8,45	8,47	1,78	1,16	3,83	6,23
MT-kNN - 6	5,64	5,13	5,56	4,94	4,56	5,33
MT-kNN - 12	9,91	8,37	8,46	5,52	4,20	8,25

Tabela 5.19: *Speedup sequencial em relação ao rainbow, e com multithreading.*

Na Tabela 5.21 temos o *speedup* da indexação do FiSH-kNN em relação aos outros. Para comparar com o Rainbow somamos o tempo de entrada do FiSH-kNN (Tabela 5.20) com o tempo de indexação. Teve *speedup* para todos datasets em relação ao Rainbow, mas apenas nas bases maiores em relação ao MT-kNN. Em relação ao GT-kNN, o overhead de usar a Thrust deixa o *speedup* bem abaixo de 1x.

Na Tabela 5.22 há o pico de memória durante as execuções de cada método. Para os métodos de CPU reportamos o uso de RAM, e os de GPU o uso da memória da GPU. Apesar de serem os mesmos dados, a estrutura das entradas do GT-kNN, MT-KNN e FiSH-kNN são maiores, fazendo com que usem mais memória que o G-KNN e Rainbow. O uso de memória no FiSH-kNN com tamanhos de grupo de consultas 20 e 50,

Dataset	Tempo
20NG	0,55
4Uni	0,27
ACM	0,25
Medline	7,64
RCV1	14,08
Reuters90	0,27

Tabela 5.20: *Tempo em segundos do processamento da entrada dos métodos kNN.*

	20NG	4Uni	ACM	Medline	RCV1	Reuters90
Rainbow	2,45	2,08	1,44	2,82	3,55	1,79
MT-kNN - 1	0,51	0,26	0,13	3,43	4,41	0,23
MT-kNN - 6	0,26	0,14	0,09	1,49	1,99	0,14
MT-kNN - 12	0,18	0,09	0,07	1,05	1,30	0,10
G-KNN	1,47	1,52	1,64	0,88	0,62	1,64
GT-kNN	0,09	0,05	0,03	0,54	0,67	0,05

Tabela 5.21: *Speedup da indexação do FiSH-kNN sobre os outros.*

naturalmente aumentam um pouco em relação ao de tamanho 1, por precisar de alocar estruturas para B consultas. Essas soluções necessitam que a base caiba por completo na memória da GPU.

	20NG	4Uni	ACM	Medline	RCV1	Reuters90
Rainbow	38	21	18	400	795	17
MT-kNN - 1	69	34	24	906	1.988	32
MT-kNN - 6	76	37	30	1.134	2.202	36
MT-kNN - 12	82	41	37	1.409	2.460	39
G-KNN	91	81	78	289	566	81
GT-kNN	150	118	104	907	1.996	116
FiSH-kNN 1B	144	112	100	902	1.991	110
FiSH-kNN 20B	147	113	103	1.027	2.105	112
FiSH-kNN 50B	153	117	107	1.222	2.289	114

Tabela 5.22: *Pico de memória em Megabytes.*

5.4.2 Análise do Uso de Grupos de Consultas

Na Tabela 5.23 temos o tempo de consulta variando o tamanho de grupo e número de GPUs do FiSH-kNN. Como podemos ver o tempo dificilmente tem uma melhora, e no caso da RCV1 piora um pouco. Apesar de ganhar tempo ao enviar várias consultas por vez, o tempo maior gasto ao preparar os vetores das consultas na CPU, faz com que grupos maiores não tenham ganhos de tempo. Ao comparar o tamanho de grupo 1 com os tamanhos 5, 20 e 50, na Tabela 5.24 podemos ver que o tamanho 5 é suficiente para obter um *speedup* de 1,1x, uma pequena melhora.

#GPUs	1			2			4		
Dataset / Group size	5	20	50	5	20	50	5	20	50
20NG	0,47	0,47	0,52	0,26	0,25	0,27	0,15	0,15	0,15
4Uni	0,42	0,41	0,38	0,24	0,23	0,21	0,15	0,14	0,14
ACM	0,34	0,32	0,32	0,17	0,17	0,17	0,10	0,10	0,09
Medline	3,50	3,42	3,61	1,67	1,66	1,68	0,84	0,83	0,83
RCV1	7,51	7,78	7,77	3,68	3,77	3,80	1,83	1,88	1,89
Reuters90	0,35	0,34	0,33	0,20	0,19	0,18	0,12	0,11	0,11

Tabela 5.23: *Tempo em milissegundos para uma consulta com variados tamanhos de grupo.*

#GPUs	1			2			4		
Dataset / Tam, grupo	5	20	50	5	20	50	5	20	50
20NG	1,16	1,16	1,07	1,11	1,14	1,07	1,13	1,14	1,08
4Uni	1,19	1,22	1,30	1,20	1,22	1,35	1,12	1,18	1,14
ACM	1,18	1,23	1,25	1,17	1,19	1,24	1,13	1,15	1,21
Medline	0,98	1,00	0,95	1,01	1,02	1,01	1,01	1,02	1,02
RCV1	0,93	0,89	0,89	0,92	0,90	0,89	0,92	0,90	0,89
Reuters90	1,22	1,29	1,30	1,13	1,18	1,22	1,08	1,18	1,15
Average	1,11	1,13	1,13	1,09	1,11	1,13	1,07	1,10	1,08

Tabela 5.24: *Speedup do uso de grupos.*

5.4.3 Análise do Tipo de Balanceamento

Na Tabela 5.25 podemos ver os tempos do FiSH-kNN ao usar o balanceamento intra-bloco e o *speedup* sobre o balanceamento inter-bloco. Como esperado, na média não há realmente diferença, pois a latência pode ser escondida entre blocos de *threads* na GPU.

	20NG	4Uni	ACM	Medline	RCV1	Reuters90
FiSH-kNN - 1	0,54	0,47	0,39	3,41	6,81	0,42
FiSH-kNN - 2	0,29	0,27	0,21	1,73	3,31	0,23
FiSH-kNN - 4	0,16	0,16	0,12	0,87	1,65	0,13
Speedup - 1	1,02	1,07	1,01	1,00	1,02	1,04
Speedup - 2	1,01	1,05	0,97	0,98	1,02	0,97
Speedup - 4	1,01	1,05	0,97	0,98	1,02	0,97

Tabela 5.25: *Tempo de consulta com balanceamento intra-bloco e speedup sobre o balanceamento inter-bloco.*

Ao usar grupos, na Tabela 5.26 temos o *speedup* do balanceamento inter-bloco sobre o intra-bloco. Algumas coleções chegam até 1,1x de *speedup* com um grupo maior, mas como visto na média da última linha, os ganhos são de no máximo 4%. Então o tipo de balanceamento não é realmente crítico para essa aplicação.

#GPUs	1			2			4		
Dataset / Tam, grupo	5	20	50	5	20	50	5	20	50
20NG	1,03	1,06	1,13	1,05	1,06	1,09	1,03	1,07	1,11
4Uni	1,09	1,08	1,02	1,08	1,09	0,98	1,10	1,04	1,07
ACM	1,01	1,01	0,99	0,99	1,01	0,98	1,01	1,02	0,98
Medline	1,06	1,04	1,10	1,00	1,00	1,01	1,00	1,00	0,99
RCV1	1,01	1,02	1,01	1,02	1,01	1,01	1,02	1,01	1,00
Reuters90	1,04	1,01	1,01	1,04	1,03	0,99	1,05	1,03	1,01
Average	1,04	1,04	1,04	1,03	1,03	1,01	1,03	1,03	1,03

Tabela 5.26: *Speedup do balanceamento inter-bloco sobre o intra-bloco, com uso de grupos.*

CAPÍTULO 6

Conclusão

Existem diversas maneiras de processar documentos, e algumas delas são aplicadas em buscas na Web, pois uma página pode ser considerada como um documento. Devido à crescente demanda de consultas por usuários, e a já existente grande quantidade de páginas, torna-se um desafio processar essas consultas de forma eficiente e atender essas demandas. Este trabalho teve como foco três problemas: *a*) Aprendizado Ativo, que consiste na seleção de um subconjunto de documentos que melhor representam um superconjunto, criando uma base de dados reduzida; *b*) *Learning to Rank*, que é a tarefa de ranquear consultas para que os documentos mais relevantes para uma busca estejam no início de uma lista; *c*) Busca Top-K, que usa o método de k-vizinhos mais próximos para calcular a similaridade entre documentos baseado em alguma métrica de distância ou similaridade. Foram propostos algoritmos paralelos para cada uma delas.

Nossos experimentos mostraram a independência do processamento dos documentos, permitindo a distribuição de diferentes documentos para diferentes processadores sem a necessidade de sincronização entre eles, e tendo ganhos de *speedups* na grande maioria dos experimentos.

Nossa solução de aprendizado ativo *multicore* PSSARP obteve *speedup* de 10.8x no melhor caso com 12 *threads* em relação ao serial, o que é próximo do ideal (12x). E nossa versão *manycore* PRSS obteve 128x de *speedup* em relação ao serial usando 1 GPU. Ao usar 4 GPUs, o *speedup* foi de 3,5x em relação à 1 GPU.

Com nossa solução *manycore* de *learning to rank* PROFL, atingimos 508x de *speedup* em relação ao serial, 9x sobre uma *baseline manycore*, e 4x sobre o PROFL com 1 GPU ao usar 4 GPUs, nos melhores casos. Também foi atingido 4x de *speedup* em relação ao PROFL sem o uso da abordagem por blocos, tanto com 1 e 4 GPUs, mostrando a eficiência dessa abordagem.

Na busca top-k nossa solução *multicore* MT-kNN obteve 10x de *speedup* no melhor caso, e serviu de *baseline* para nossa proposta *manycore* FiSH-kNN, que é uma extensão do GT-kNN. O FiSH-kNN obteve 2,7x de *speedup* em relação ao GT-kNN, 17x sobre uma versão serial, 4x sobre o MT-kNN, e 4x ao usar 4 GPUs em relação a 1 GPU, nos melhores casos.

Nestas três soluções *manycore* (PRSS, PROFL, FiSH-kNN), a implementação com multi-GPU foi feita com a simples divisão de documentos entre cada uma. A nossa contribuição com o PRSS permitirá que todos trabalhos relacionados que usavam o SSARP, acelerem suas implementações e ainda possam avaliar seus algoritmos com bases de milhares de consultas, viabilizadas somente com o uso do PRSS. Outra contribuição, é a paralelização total do sistema de ranqueamento sob demanda com regras de associação, pois a base de dados reduzida viabiliza o aspecto sob demanda. O PROFL acelera esse aspecto ainda mais, com sua abordagem escalável por bloco de *threads* e uso de *cache*.

Com o FiSH-kNN pudemos acelerar a busca dos Top-K vizinhos mais próximos, e avaliamos um balanceamento de carga intra-bloco, que mostrou ser apenas até 4% inferior que o balanceamento inter-bloco. também avaliamos o processamento de documentos por grupos, para reduzir a sobrecarga de transferência de memória, e nos experimentos um grupo de tamanho 5 já é suficiente para ter um ganho de 10%. Foram ganhos modestos, com exceção da multi-GPU que obteve *speedup* ideal nas bases de dados maiores, mas todo trabalho relacionado que necessita executar o kNN várias vezes, como geração de *metafeatures* e sistemas de recomendação, e em bases de alta esparsidade e dimensionalidade, terá seu desempenho aumentado.

6.1 Trabalhos Futuros

Definimos os seguintes pontos como sugestões para trabalhos futuros:

- Verificar a acurácia do ranqueamento com as bases reduzidas de variados números de partições e tamanhos de regra, usando o PROFL e métodos de *learning to rank* estado da arte.
- Reduzir o tempo de processamento do PROFL com uma estratégia de particionamento vertical como no PRSS, e checar o efeito na acurácia.
- Implementar o FiSH-kNN com um índice invertido dividido entre as GPUs, para bases que não caibam em uma GPU.
- Implementar podas de espaço de busca no FiSH-kNN para reduzir o tempo de cálculo de distâncias.
- Adaptar o FiSH-kNN para fazer clusterização k-*means* (k-médias) e filtragem colaborativa em bases de sistemas de recomendação.
- Juntar as versões *multicore* e *manycore* dos algoritmos, para que ambas GPU e CPU trabalhem em paralelo, e com uma carga balanceada.

6.2 Artigos Publicados

Nosso trabalho do PROFL teve um artigo publicado no XVII Simpósio em Sistemas Computacionais de Alto Desempenho (WSCAD), intitulado "A Fast and Scalable Manycore Implementation for an On-Demand Learning to Rank Method".

Referências Bibliográficas

[1] ADENIYI, D.; WEI, Z.; YONGQUAN, Y. **Automated web usage data mining and recommendation system using k-nearest neighbor (knn) classification method**. *Applied Computing and Informatics*, 12(1):90–108, 2016.

[2] AGRAWAL, R.; IMIELIŃSKI, T.; SWAMI, A. **Mining association rules between sets of items in large databases**. In: *ACM SIGMOD Record*, volume 22, p. 207–216. ACM, 1993.

[3] AGRAWAL, R.; SRIKANT, R.; OTHERS. **Fast algorithms for mining association rules**. In: *Proc. 20th int. conf. very large data bases, VLDB*, volume 1215, p. 487–499, 1994.

[4] ATREAS, N.; KARANIKAS, C. **A fast pattern matching algorithm based on prime numbers and hashing approximation**. *NATO SECURITY THROUGH SCIENCE SERIES D-INFORMATION AND COMMUNICATION SECURITY*, 12:118, 2007.

[5] BUCKLES, B. P.; LYBANON, M. **Algorithm 515: Generation of a vector from the lexicographical index [g6]**. *ACM Transactions on Mathematical Software (TOMS)*, 3(2):180–182, 1977.

[6] BURGES, C.; SHAKED, T.; RENSHAW, E.; LAZIER, A.; DEEDS, M.; HAMILTON, N.; HULLENDER, G. **Learning to rank using gradient descent**. In: *Proceedings of the 22nd international conference on Machine learning*, p. 89–96. ACM, 2005.

[7] CANUTO, S.; GONÇALVES, M.; SANTOS, W.; ROSA, T.; MARTINS, W. **An efficient and scalable metafeature-based document classification approach based on massively parallel computing**. In: *Proceedings of the 38th International ACM SIGIR Conference on Research and Development in Information Retrieval*, p. 333–342. ACM, 2015.

[8] CHAPELLE, O.; CHANG, Y.; LIU, T.-Y. **Future directions in learning to rank**. In: *Proceedings of the 2010 International Conference on Yahoo! Learning to Rank Challenge - Volume 14*, YLRC'10, p. 91–100. JMLR.org, 2010.

[9] DE ALMEIDA, H. M.; GONÇALVES, M. A.; CRISTO, M.; CALADO, P. **A combined component approach for finding collection-adapted ranking functions based on genetic programming**. In: *Proceedings of the 30th annual international ACM SIGIR conference on Research and development in information retrieval*, p. 399–406. ACM, 2007.

[10] DE SOUSA, D. X.; ROSA, T. C.; MARTINS, W. S.; SILVA, R.; GONÇALVES, M. A. **Improving on-demand learning to rank through parallelism**. In: *Web Information Systems Engineering-WISE 2012*, p. 526–537. Springer, 2012.

[11] DJENOURI, Y.; BENDJOUDI, A.; MEHDI, M.; NOUALI-TABOUDJEMAT, N.; HABBAS, Z. **Gpu-based bees swarm optimization for association rules mining**. *The Journal of Supercomputing*, 71(4):1318–1344, 2015.

[12] GARCIA, V.; DEBREUVE, E.; BARLAUD, M. **Fast k nearest neighbor search using gpu**. In: *Computer Vision and Pattern Recognition Workshops, 2008. CVPRW'08. IEEE Computer Society Conference on*, p. 1–6. IEEE, 2008.

[13] GIONIS, A.; INDYK, P.; MOTWANI, R. **Similarity search in high dimensions via hashing**. In: *Proceedings of the 25th International Conference on Very Large Data Bases*, VLDB '99, p. 518–529, San Francisco, CA, USA, 1999. Morgan Kaufmann Publishers Inc.

[14] HAN, J.; KAMBER, M.; PEI, J. **Data Mining: Concepts and Techniques 3rd Edition**. Morgan Kaufmann Publishers Inc., 2012.

[15] HARRIS, M. **How to overlap data transfers in cuda c/c++**, Dec. 2012. https://devblogs.nvidia.com/parallelforall/how-overlap-data-transfers-cuda-cc/, acessado em junho de 2017.

[16] HARRIS, M.; OTHERS. **Optimizing parallel reduction in cuda**. *NVIDIA Developer Technology*, 2(4), 2007.

[17] HASTIE, T.; TIBSHIRANI, R.; FRIEDMAN, J. **The elements of statistical learning: data mining, inference and prediction**. Springer, 2 edition, 2009.

[18] JIN, J.; CAI, X.; LAI, G.; LIN, X. **Gpu-accelerated parallel algorithms for linear ranksvm**. *The Journal of Supercomputing*, 71(11):4141–4171, 2015.

[19] KATO, K.; HOSINO, T. **Solving k-nearest neighbor problem on multiple graphics processors**. In: *Proceedings of the 2010 10th IEEE/ACM International Conference on Cluster, Cloud and Grid Computing*, p. 769–773. IEEE Computer Society, 2010.

[20] LONG, B.; BIAN, J.; CHAPELLE, O.; ZHANG, Y.; INAGAKI, Y.; CHANG, Y. **Active learning for ranking through expected loss optimization**. *Knowledge and Data Engineering, IEEE Transactions on*, 27(5):1180–1191, 2015.

[21] LUKAČ, N.; ŽALIK, B. **Fast approximate k-nearest neighbours search using gpgpu**. In: *GPU Computing and Applications*, p. 221–234. Springer, 2015.

[22] MANNING, C. D.; RAGHAVAN, P.; SCHÜTZE, H. **Introduction to Information Retrieval**. Cambridge University Press, New York, NY, USA, 2008.

[23] MEHROTRA, R.; YILMAZ, E. **Representative & informative query selection for learning to rank using submodular functions**. In: *Proceedings of the 38th International ACM SIGIR Conference on Research and Development in Information Retrieval*, p. 545–554. ACM, 2015.

[24] NVIDIA. **NVIDIA's Next Generation CUDA TM Compute Architecture: Kepler GK110**, 2012. V1.0. https://www.nvidia.com/content/PDF/kepler/NVIDIA-Kepler-GK110-Architecture-Whitepaper.pdf, acessado em junho de 2017.

[25] NVIDIA. **CUDA Parallel Reduction**, Aug. 2014. http://docs.nvidia.com/cuda/cuda-samples/index.html#cuda-parallel-reduction, acessado em junho de 2017.

[26] NVIDIA. **CUDA C PROGRAMMING GUIDE**, Aug. 2015. 7.5 Edition. http://docs.nvidia.com/cuda/cuda-c-programming-guide/, acessado em junho de 2017.

[27] PAGE, L.; BRIN, S.; MOTWANI, R.; WINOGRAD, T. **The pagerank citation ranking: Bringing order to the web.** Technical report, Stanford InfoLab, 1999. DOI: 10.1.1.31.1768.

[28] QIN, T.; LIU, T.-Y.; XU, J.; LI, H. **Letor: A benchmark collection for research on learning to rank for information retrieval**. *Information Retrieval*, 13(4):346–374, 2010.

[29] RICCI, F.; ROKACH, L.; SHAPIRA, B.; KANTOR, P. B. **Recommender systems handbook**. Springer, 2015.

[30] ROCHA, L.; RAMOS, G.; CHAVES, R.; SACHETTO, R.; MADEIRA, D.; VIEGAS, F.; ANDRADE, G.; DANIEL, S.; GONÇALVES, M.; FERREIRA, R. **G-knn: an efficient document classification algorithm for sparse datasets on gpus using knn**. In: *Proceedings of the 30th Annual ACM Symposium on Applied Computing*, p. 1335–1338. ACM, 2015.

[31] SCHMIDBERGER, G.; FRANK, E. **Unsupervised discretization using tree-based density estimation**. In: *European Conference on Principles of Data Mining and Knowledge Discovery*, p. 240–251. Springer, 2005.

[32] SENGUPTA, S.; HARRIS, M.; GARLAND, M.; OWENS, J. D. **Efficient parallel scan algorithms for many-core gpus**. *Scientific Computing with Multicore and Accelerators*, p. 413–442, 2011.

[33] SILVA, R.; GONÇALVES, M. A.; VELOSO, A. **Rule-based active sampling for learning to rank**. In: *Machine Learning and Knowledge Discovery in Databases*, p. 240–255. Springer, 2011.

[34] SILVA, R. M.; GONÇALVES, M. A.; VELOSO, A. **A two-stage active learning method for learning to rank**. *Journal of the Association for Information Science and Technology*, 65(1):109–128, 2014.

[35] SISMANIS, N.; PITSIANIS, N.; SUN, X. **Parallel search of k-nearest neighbors with synchronous operations.** In: *HPEC*, p. 1–6. IEEE, 2012.

[36] TEODORO, G.; MARIANO, N.; MEIRA, W.; FERREIRA, R. **Tree projection-based frequent itemset mining on multicore cpus and gpus**. In: *Computer Architecture and High Performance Computing (SBAC-PAD), 2010 22nd International Symposium on*, p. 47–54. IEEE, 2010.

[37] TYREE, S.; WEINBERGER, K. Q.; AGRAWAL, K.; PAYKIN, J. **Parallel boosted regression trees for web search ranking**. In: *Proceedings of the 20th international conference on World wide web*, p. 387–396. ACM, 2011.

[38] VELOSO, A. A.; ALMEIDA, H. M.; GONÇALVES, M. A.; MEIRA JR, W. **Learning to rank at query-time using association rules**. In: *Proceedings of the 31st annual international ACM SIGIR conference on Research and development in information retrieval*, p. 267–274. ACM, 2008.

[39] WANG, B.; WU, T.; YAN, F.; LI, R.; XU, N.; WANG, Y. **Rankboost acceleration on both nvidia cuda and ati stream platforms**. In: *Parallel and Distributed Systems (ICPADS), 2009 15th International Conference on*, p. 284–291. IEEE, 2009.

[40] WANG, S.; WU, Y.; GAO, B. J.; WANG, K.; LAUW, H. W.; MA, J. **A cooperative coevolution framework for parallel learning to rank**. *Knowledge and Data Engineering, IEEE Transactions on*, 27(12):3152–3165, 2015.

[41] YUE, Y.; FINLEY, T.; RADLINSKI, F.; JOACHIMS, T. **A support vector method for optimizing average precision**. In: *Proceedings of the 30th annual international ACM*

SIGIR conference on Research and development in information retrieval, p. 271–278. ACM, 2007.

[42] ZAKI, M. J.; MEIRA JR, W. **Data Mining and Analysis: Fundamental Concepts and Algorithms**. Cambridge University Press, 2014.

[43] ZHANG, F.; ZHANG, Y.; BAKOS, J. **Gpapriori: Gpu-accelerated frequent itemset mining**. In: *Cluster Computing (CLUSTER), 2011 IEEE International Conference on*, p. 590–594. IEEE, 2011.

[44] ZHOU, J.; GUO, Q.; JAGADISH, H. V.; LUAN, W.; TUNG, A. K. H.; YANG, Y.; ZHENG, Y. **Generic inverted index on the GPU**. *CoRR*, abs/1603.08390, 2016.