Soluções aproximadas para algoritmos escaláveis de mineração de dados em domínios de dados complexos usando GPGPU

Alexander Victor Ocsa Mamani

SERVIÇO DE PÓS-GRADUAÇÃO DO ICMC-USP

Data de Depósito: 27/10/2011

Assinatura:________________________

Soluções aproximadas para algoritmos escaláveis de mineração de dados em domínios de dados complexos usando GPGPU

Alexander Victor Ocsa Mamani

Orientadora: **Profa. Dra. Elaine Parros Machado de Sousa**

Dissertação apresentada ao Instituto de Ciências Matemáticas e de Computação - ICMC-USP, como parte dos requisitos para obtenção do título de Mestre em Ciências - Ciências de Computação e Matemática Computacional. *VERSÃO REVISADA*

USP – São Carlos
Outubro de 2011

Ficha catalográfica elaborada pela Biblioteca Prof. Achille Bassi
e Seção Técnica de Informática, ICMC/USP,
com os dados fornecidos pelo(a) autor(a)

M263s

Mamani, Alexander Victor Ocsa
Soluções aproximadas para algoritmos escaláveis de dados em domínios de dados complexos usando GPGPU / Alexander Victor Ocsa Mamani; orientadora Elaine Parros Machado de Sousa -- São Carlos, 2011.
90 p.

Dissertação (Mestrado - Programa de Pós-Graduação em Ciências de Computação e Matemática Computacional) -- Instituto de Ciências Matemáticas e de Computação, Universidade de São Paulo, 2011.

1. busca ao vizinho mais próximo . 2. mineração de dados. 3. descoberta de motifs. 4. GPGPU. 5. CUDA. I. Sousa, Elaine Parros Machado de, orient. II. Título.

Agradecimentos

A minha orientadora, Profa. Dra. Elaine Parros Machado de Sousa, sempre presente e atenciosa. Por todos os anos de orientação, dos trabalhos de pesquisa à conclusão desta dissertação de mestrado. Pelos desafios que me proporcionou e, principalmente, por sua confiança.

Aos meus pais, Claudio e Cecilia, pela educação, amor e carinho. A minha irmã Ruth, por ser a pessoa que ainda consegue me surpreender. E a todas as pessoas que, direta ou indiretamente, contribuíram para que eu chegasse até aqui.

Ao Instituto de Ciências Matemáticas e de Computação (ICMC), pela estrutura acadêmica que tornou possível o desenvolvimento deste trabalho. Aos professores e colegas do Grupo de Bases de Dados e Imagens (GBDI-ICMC-USP).

Ao Conselho Nacional de Desenvolvimento Científico e Tecnológico (CNPq) e à Fundação de Amparo à Pesquisa do Estado de São Paulo (FAPESP), pero auxílio financeiro.

Resumo

A crescente disponibilidade de dados em diferentes domínios tem motivado o desenvolvimento de técnicas para descoberta de conhecimento em grandes volumes de dados complexos. Trabalhos recentes mostram que a busca em dados complexos é um campo de pesquisa importante, já que muitas tarefas de mineração de dados, como classificação, detecção de agrupamentos e descoberta de *motifs*, dependem de algoritmos de busca ao vizinho mais próximo. Para resolver o problema da busca dos vizinhos mais próximos em domínios complexos muitas abordagens determinísticas têm sido propostas com o objetivo de reduzir os efeitos da "maldição da alta dimensionalidade". Por outro lado, algoritmos probabilísticos têm sido pouco explorados. Técnicas recentes relaxam a precisão dos resultados a fim de reduzir o custo computacional da busca. Além disso, em problemas de grande escala, uma solução aproximada com uma análise teórica sólida mostra-se mais adequada que uma solução exata com um modelo teórico fraco. Por outro lado, apesar de muitas soluções exatas e aproximadas de busca e mineração terem sido propostas, o modelo de programação em CPU impõe restrições de desempenho para esses tipos de solução. Uma abordagem para melhorar o tempo de execução de técnicas de recuperação e mineração de dados em várias ordens de magnitude é empregar arquiteturas emergentes de programação paralela, como a arquitetura CUDA. Neste contexto, este trabalho apresenta uma proposta para buscas kNN de alto desempenho baseada numa técnica de *hashing* e implementações paralelas em CUDA. A técnica proposta é baseada no esquema LSH, ou seja, usa-se projeções em subespaços. O LSH é uma solução aproximada e tem a vantagem de permitir consultas de custo sublinear para dados em altas dimensões. Usando implementações massivamente paralelas melhora-se tarefas de mineração de dados. Especificamente, foram desenvolvidos soluções de alto desempenho para algoritmos de descoberta de *motifs* baseados em implementações paralelas de consultas kNN. As implementações massivamente paralelas em CUDA permitem executar estudos experimentais sobre grandes conjuntos de dados reais e sintéticos. A avaliação de desempenho realizada neste trabalho usando GeForce GTX470 GPU resultou em um aumento de desempenho de até 7 vezes, em média sobre o estado da arte em buscas por similaridade e descoberta de *motifs*.

Palavras Chave: busca ao vizinho mais próximo, busca por similaridade aproximada, projeção aleatória, mineração de dados, descoberta de *motifs*, dados complexos, GPGPU, CUDA

Abstract

The increasing availability of data in diverse domains has created a necessity to develop techniques and methods to discover knowledge from huge volumes of complex data, motivating many research works in databases, data mining and information retrieval communities. Recent studies have suggested that searching in complex data is an interesting research field because many data mining tasks such as classification, clustering and motif discovery depend on nearest neighbor search algorithms. Thus, many deterministic approaches have been proposed to solve the nearest neighbor search problem in complex domains, aiming to reduce the effects of the well-known "curse of dimensionality". On the other hand, probabilistic algorithms have been slightly explored. Recently, new techniques aim to reduce the computational cost relaxing the quality of the query results. Moreover, in large-scale problems, an approximate solution with a solid theoretical analysis seems to be more appropriate than an exact solution with a weak theoretical model. On the other hand, even though several exact and approximate solutions have been proposed, single CPU architectures impose limits on performance to deliver these kinds of solution. An approach to improve the runtime of data mining and information retrieval techniques by an order-of-magnitude is to employ emerging many-core architectures such as CUDA-enabled GPUs. In this work we present a massively parallel kNN query algorithm based on hashing and CUDA implementation. Our method, based on the LSH scheme, is an approximate method which queries high-dimensional datasets with sub-linear computational time. By using the massively parallel implementation we improve data mining tasks, specifically we create solutions for (soft) real-time time series motif discovery. Experimental studies on large real and synthetic datasets were carried out thanks to the highly CUDA parallel implementation. Our performance evaluation on GeForce GTX 470 GPU resulted in average runtime speedups of up to 7x on the state-of-art of similarity search and motif discovery solutions.

Keywords: nearest neighbor search, similarity search, random projection, data mining, motif-discovery, complex data, GPGPU, CUDA.

Sumário

Agradecimentos 5
Resumo i
Abstract iii
Sumário v
Siglas vii
Lista de Símbolos ix
Lista de Figuras xi
Lista de Tabelas xiii
Lista de Algoritmos xv

1 Introdução **1**
1.1 Considerações Iniciais 1
1.2 Definição do Problema e Motivação 2
1.3 Objetivos 4
1.4 Principais Contribuições 4
1.5 Organização do Trabalho 5

2 Consulta por similaridade sobre dados complexos **7**
2.1 Considerações Iniciais 7
2.2 Domínio dos Dados 9
2.2.1 Espaços Métricos 9
2.2.2 Espaços Multidimensionais 10
2.3 Consultas por Similaridade 11
2.4 A Maldição da Alta Dimensionalidade 14
2.5 Abordagens para a Solução Aproximada 14
2.5.1 Space-Filling Curves 15
2.5.2 *Locality Sensitive Hashing* 16
2.5.3 HashFile 20
2.6 Considerações Finais 24

3 Mineração de Séries Temporais **25**
3.1 Considerações Iniciais 25
3.2 Representação de Séries Temporais 26
3.3 Medidas de Similaridade em Séries Temporais 28
3.4 Consulta por Similaridade em Séries Temporais 30
3.5 KDD e Mineração de Dados 31
3.6 Descoberta de *motifs* 32
3.6.1 Algoritmo de força bruta 34

3.6.2 Algoritmo MK 35
3.6.3 Descoberta de Motifs usando *Random Projection* 36
3.6.4 *MrMotif* 37
3.7 Considerações Finais 38

4 GPGPU em CUDA 39
4.1 Considerações Iniciais 39
4.2 Arquitetura CUDA 40
4.2.1 Modelo de Programação em CUDA 41
4.2.2 Modelo de Memória 43
4.3 Restrições de implementação 44
4.4 Exemplo em CUDA 45
4.5 Aplicações 48
4.5.1 Operações de processamento de Consultas em Banco de Dados utilizando GPUs 48
4.5.2 Tarefas de Mineração de Dados utilizando GPUs 48
4.6 Considerações Finais 49

5 Soluções Aproximadas de Busca por Similaridade e Descoberta de *Motifs* usando GPGPU 51
5.1 Considerações Iniciais 51
5.2 Visão Geral do método de indexação CUDA-LSH 52
5.2.1 Organização da Estrutura 52
5.2.2 Construção do CUDA-LSH 53
5.2.3 Consultas kNN aproximada no CUDA-LSH 55
5.2.4 Consultas AllkNN aproximada no CUDA-LSH 57
5.3 Descoberta de *motifs* em CUDA 59
5.3.1 Descoberta de *motifs* baseado no algoritmo CUDA-AllkNN 59
5.3.2 *Random Projection* em CUDA 60
5.4 Considerações Finais 62

6 Experimentos 63
6.1 Considerações Iniciais 63
6.2 Materiais e Métodos 64
6.3 Resultados e Discussão 69
6.3.1 Consultas por abrangência exata e aproximada 69
6.3.2 Consultas kNN e AllkNN aproximadas 74
6.3.3 Descoberta de *motifs* 77
6.4 Considerações Finais 80

7 Conclusão 81
7.1 Considerações Finais 81
7.2 Principais Contribuições 83
7.3 Propostas de Trabalhos Futuros 83

Referências Bibliográficas 90

Siglas

CUDA	*Compute Unified Device Architecture*.
CPU	*Central Processing Unit*.
DTW	*Dynamic Time Warping*.
GEMINI	*GEneric Multimedia INdexIng*.
GBDI-ICMC-USP	Grupo de Bases de Dados e Imagens Instituto de Ciências Matemáticas e de Computação - USP.
GPGPU	*General Purpose computing on Graphics Processing Unit*.
GPU	*Graphics Processing Unit*.
iSAX	*indexable Symbolic Aggregate Approximation*.
KDD	*Knowledge Discovery in Databases*.
LSH	*Locality Sensitive Hashing*.
MA	Método de Acesso.
MAM	Método de Acesso Métrico.
MAE	Método de Acesso Espacial.
NCD	Número de Calculos de Distância.
SAX	*Symbolic Aggregate Approximation*.
TT	Tempo Total.

Lista de Símbolos

S	conjunto de dados
N	número de objetos no conjunto de dados
$d()$	função de distância
q	objeto de consulta
r	raio de consulta
k	número de objetos mais próximos em uma consulta kNN
ε	erro relativo na busca por similaridade aproximada
$(1+\varepsilon)$	o fator de aproximação
$Rq(q,r)$	consulta por abrangência
$kNN(q,k)$	consulta aos k-vizinhos mais próximos
$AllkNN(k)$	consulta a todos os k-vizinhos mais próximos
$(1+\epsilon)-Rq(q,r)$	consulta por abrangência aproximada
$(1+\epsilon)-kNN(q,k)$	consulta aproximada aos k-vizinhos mais próximos
$(1+\epsilon)-AllkNN(k)$	consulta aproximada a todos os k-vizinhos mais próximos
$h()$	função *hash*
$H=\{h_1(),...,h_m()\}$	um conjunto de funções *hash* de tamanho m
L	número de tabelas *hash*
m	número de funções *hash* (comprimento da projeção *hash*)
w	largura de quantização para os *buckets*
τ	número de *buckets* ou $probes$ a serem explorados
λ	intervalo de consulta para buscas $(1+\epsilon)-kNN(q,k)$
C	capacidade do *bucket*
$T=(T_1,T_2,...,T_n)$	série temporal

Lista de Figuras

2.1 Unificação do modelo de consulta por similaridade. (a) Os Métodos de Acesso Métricos (MAMs). (b) *Locality Sensitive Hashing* (LSH). 9
2.2 (a) Formas geométricas que ilustram a delimitação da região de busca de acordo com a métrica L_p utilizada. (b) Exemplo de consulta por abrangência para diferentes métricas da família L_p. 10
2.3 Exemplo de consulta por abrangência. 11
2.4 Exemplo de consulta por vizinhos mais próximos. 12
2.5 Exemplo de consultas AllkNN para um conjunto de dados bidimensional. 12
2.6 Exemplo de consulta por abrangência aproximada. 13
2.7 Interpretação geométrica de uma busca aproximada 5-NN. (a) busca exata. (b) busca aproximada. O objeto de consulta é o ponto central q, os pontos pretos compõem o conjunto solução. 14
2.8 Alguns exemplos de *2D space-filling curves*. a) Hilbert; b) Lebesgue ou “Z-order”; c) Sierpinski. As três iterações de cada curva são apresentadas [Valle, 2008]. 15
2.9 O problema com o efeito de fronteira nas *space-filling curves*. Os pontos no centro do espaço estão mais afastados na curva do que os pontos do quadrante inferior esquerdo [Valle, 2008]. 16
2.10 Interpretação geométrica da projeção dos vetores $\vec{p_1}$ e $\vec{p_2}$ em LSH. 17
2.11 Estrutura do nó no HashFile. Figura adaptada de [Zhang et al., 2011]. 22
2.12 Busca kNN aproximada no nó do HashFile. Figura adaptada de [Zhang et al., 2011]. . . 24

3.1 Hierarquia de representações de séries temporais. Os nós folha referem-se à representação atual, e os nós internos referem-se à classificação da abordagem [Lin et al., 2003]. 26
3.2 A série temporal (linha preta), já normalizada, é discretizada, obtendo-se uma representação PPA (linha cinza grossa). Depois os coeficientes PAA em letras (em negrito) são mapeados a uma representação discreta. Neste exemplo, a série é discretizada para a palavra **cbccbaab** [Lin et al., 2007]. 27
3.3 A série temporal T convertido em palavras SAX. (a) com cardinalidade 4 $\{11, 11, 01, 00\}$ (b) com cardinalidade 2 $\{1, 1, 0, 0\}$ [Shieh e Keogh, 2008]. 28
3.4 Apesar das duas séries terem formas similares, elas não estão alinhadas no eixo do tempo. A distância Euclidiana gera uma medida de dissimilaridade pessimista, já o DTW produz uma medida de dissimilaridade mais intuitiva devido aos alinhamentos não-lineares [Keogh, 2002]. 29
3.5 Matriz de alinhamento para o cálculo do DTW. A área em cinza representa uma janela de alinhamento [Keogh, 2002]. 29
3.6 Etapas do processo KDD. Figura adaptada de [Fayyad et al., 1996]. 31

3.7 Representação esquemática do *Motif*. 33
3.8 As séries temporais que definem o *motif* na Figura 3.7 compartilhan a mesma palavra SAX. 37

4.1 A arquitetura CUDA mostra um conjunto de processadores de unidades programáveis de uma NVIDIA GeForce GTX 8800. Figura adaptada de [Kirk e Hwu, 2010]. 41
4.2 Execução de um programa em CUDA. Figura adaptada de [Kirk e Hwu, 2010]. 42
4.3 Organização dos *threads* em CUDA. Figura adaptada de [Kirk e Hwu, 2010]. 43
4.4 Modelo de memória em CUDA. Figura adaptada de [Kirk e Hwu, 2010]. 44
4.5 Disposição bidimensional de uma coleção de blocos e *threads* utilizado no Algoritmo 4.4. 47

5.1 Estrutura lógica do índice CUDA-LSH. 53
5.2 Estrutura do índice CUDA-LSH em suas etapas distintas de construção (a) Projeção de todos os objetos (b) Ordenação em ordem crescente. 54
5.3 Etapas do processo de indexação paralela do CUDA-LSH. 55
5.4 Representação das projeções de uma consulta por similaridade aproximada no CUDA-LSH em (a), em (b) verifica-se a sua estrutura lógica correspondente. 56
5.5 Os *clusters* candidatos a *motif* utilizando os vizinhos mais próximos. 59
5.6 Pipeline de consultas kNN paralelas para a identificação dos *motifs*. 60

6.1 Comparação de resultados em consultas por abrangência (Rq) usando a média do número de cálculos de distância (primeira linha) e tempo de resposta (segunda linha) para os conjuntos de dados SYNT16 (primeira coluna), SYNT64 (segunda coluna) e SYNT256 (terceira coluna). 70
6.2 Comparação de resultados em consultas por abrangência (Rq) usando a média do número de cálculos de distância (primeira linha) e tempo de resposta (segunda linha) para os conjuntos de dados MNIST (primeira coluna), COLOR (segunda coluna) e AUDIO (terceira coluna). 71
6.3 Comparação de resultados para consultas por abrangência. Mostra-se o número de cálculos de distância (primeira linha), o tempo de resposta (segunda linha) e o *recall* (terceira linha). 73
6.4 Comparação das taxas de error (*error ratio*) para consultas kNN nos conjuntos de dados COLOR (L_1) e MNIST (L_2). 75
6.5 Comparação do tempo total de indexação e consulta AllkNN para o método CUDA-LSH em suas diferentes configurações. 76
6.6 Comparação de consultas AllkNN conforme o tamanho do conjunto de dados aumenta. O tempo total de indexação e consulta para o conjunto de dados SYNT32. 76
6.7 Comparação dos tempos de execução dos algoritmos MK e *CUDA-TopKMotif* para a identificação do *Pair-Motif*. 77
6.8 Comparação do tempo de execução para a identificação dos *Top-10 Motifs* utilizando os algoritmos *CUDA-TopKMotifs*, *CUDA-RandomProjection*, *MrMotif* e *Random Projection*. 78
6.9 Comparação do tempo de execução na identificação dos *Top-10 Motifs* conforme o tamanho do conjunto de dados RWALK cresce. 79
6.10 Comparação dos raios dos *cluster* para os *Top-10 Motifs* encontrados pelos algoritmos *CUDA-TopKMotifs*, *CUDA-RandomProjection* e *MrMotif*. 80

Lista de Tabelas

3.1 Trabalhos recentes em indexação de séries temporais. 31

6.1 Especificações da CPU e GPU assim como os compiladores utilizados nos experimentos. 66
6.2 Parâmetros LSH para os conjuntos de dados SYNT16, SYNT64, SYNT256, MNIST e AUDIO. 69
6.3 Comparação do número de cálculos de distância (NCD) para os métodos LSH. Resultados para os conjuntos de dados COLOR (32-D), MNIST (50-D), AUDIO (190-D) e SYNT256 (265-D). 71
6.4 Comparação do tempo de resposta (TT) para os métodos LSH. Resultados para os conjuntos de dados COLOR (32-D), MNIST (50-D), AUDIO (190-D) e SYNT256 (265-D). 71
6.5 Comparação de acurácia (*recall*) e do número de cálculos de distância (NCD) para os conjuntos de dados COLOR (32-D), MNIST (50-D) e AUDIO (190-D). 72
6.6 Comparação do uso da memória pelo LSH, LSH Multi-probe, LSH Multi-level, Slim-Tree, e DF-Tree em *megabytes*. 73
6.7 Comparação do tempo de consulta, *recall*, *error ratio* e uso de espaço do índice para consultas aproximadas 100-NN. Apresenta-se os resultados para os conjuntos de dados COLOR (L_1) e MNIST (L_2). 75

Lista de Algoritmos

2.1 Algoritmo de construção para LSH . . . 18
2.2 Consulta por abrangência aproximada usando LSH . . . 18
2.3 Consulta kNN aproximada usando HashFile . . . 23
3.1 Algoritmo de força bruta para a procura exata dos *TopK-Motifs* . . . 34
3.2 Algoritmo de força bruta para a procura exata do *Pair-Motif* . . . 34
3.3 Algoritmo MK para a procura exata do *Pair-Motif* . . . 35
3.4 Algoritmo baseado em projeção aleatória para a procura aproximada do *TopK-Motifs* . . . 36
3.5 Algoritmo *MrMotif* para a procura aproximada do *TopK-Motifs* . . . 38
4.1 Algoritmo CUDA-ClosestPair (versão 1) . . . 45
4.2 *Closest Pair* na CPU . . . 46
4.3 *Closest Pair* na GPU . . . 46
4.4 Algoritmo CUDA-ClosestPair (versão 2) . . . 47
5.1 Construção do índice CUDA-LSH . . . 55
5.2 kNN no CUDA-LSH . . . 57
5.3 CUDA-AllkNN no CUDA-LSH . . . 58
5.4 Algoritmo CUDA TopK-Motifs . . . 60
5.5 Algoritmo CUDA *Random Projection* (versão 1) . . . 61
5.6 Algoritmo CUDA *Random Projection* (versão 2) . . . 62

CAPÍTULO 1

Introdução

1.1 Considerações Iniciais

A crescente disponibilidade de dados em diferentes domínios, tais como a multimídia (vídeo, imagens, CAD), a internet (XML), a biologia (sequências de DNA) e a meteorologia (séries temporais climáticas), criou a necessidade de desenvolvimento de técnicas e métodos capazes de descobrir conhecimento em grandes volumes de dados complexos, motivando diversos trabalhos nas áreas de banco de dados, mineração de dados e recuperação de informação.

Atualmente, muitos desses trabalhos são voltados para aplicações em web, finanças, biologia, entre outras, em que o desempenho é um fator essencial. Nesses cenários, nem todas as aplicações precisam de uma resposta exata, pois a velocidade da consulta torna-se mais importante do que a precisão dos resultados [Liu et al., 2004, Liu et al., 2006]. Além disso, em problemas de grande escala, uma solução aproximada com uma análise teórica sólida mostra-se mais adequada do que uma solução exata com um modelo teórico fraco [Kulis e Grauman, 2009, Wang et al., 2010].

A representação, organização e, em particular, a mineração de dados complexos não são tarefas triviais. Além disso, a utilização de critérios de similaridade é particularmente comum nessas tarefas, já que a similaridade é um conceito intuitivo para a comparação de objetos complexos. Nesse contexto, diversos trabalhos mostram que a busca em dados complexos é um campo de pesquisa importante na área de mineração de dados. De modo específico, muitos algoritmos de mineração, como classificação, identificação de agrupamentos e descoberta de *motifs* (isto é, os padrões mais frequentes nos dados), dependem de algoritmos de busca ao vizinho mais próximo. Soluções exatas para resolver esse problema vêm sendo estudadas há vários anos, enquanto soluções aproximadas e probabilísticas têm sido pouco exploradas.

Na área de mineração de dados, diversos algoritmos de aprendizado desenvolvidos para tarefas supervisionadas (como a classificação) e não supervisionadas (como a identificação de agrupamentos) têm sido utilizados de maneira bem sucedida em vários domínios de aplicação. No entanto, grande parte desses algoritmos têm como desvantagem a dificuldade com o uso decorrente da dependência de

parâmetros de entrada.

Numa abordagem mais simples, algumas técnicas de mineração de dados usam o conceito de similaridade, mais especificamente os k-vizinhos mais próximos (kNN - *k-nearest neighbors*), como parte do processo de aprendizagem [Keogh, 2003, Liu et al., 2004, Liu et al., 2006].

Muitos dos métodos baseados em kNN precisam de pouco ou nenhum processo de ajuste de parâmetros, mas são limitados por sua complexidade computacional, especialmente para consultas em grandes volumes de dados complexos [Liu et al., 2004]. Além da grande quantidade de cálculos de distância, em domínios de dados complexos, os conjuntos de objetos geralmente possuem alta dimensionalidade, o que resulta em valores de distância muito próximos para a maioria dos pares de objetos – isso é chamado de "maldição da alta dimensionalidade". Desse modo, considerando que muitas tarefas de mineração de dados utilizam abordagens baseadas na similaridade dos objetos, acelerar o processo de busca kNN é um aspecto importante.

Por outro lado, embora muitas soluções exatas e aproximadas de busca e mineração tenham sido propostas, o modelo de programação em arquiteturas de CPU impõe limitações no desempenho dessas soluções. Além disso, muitas das soluções simples e gerais de recuperação e mineração de dados são consideradas impraticáveis em contextos reais devido a seu alto custo computacional. Exemplo para isso são as problemáticas referentes à busca a todos os k-vizinhos mais próximos e a procura dos K padrões mais frequentes [Metwally et al., 2005]. Uma abordagem para atacar esses problemas e melhorar o tempo de execução de técnicas de recuperação e mineração de dados, em várias ordens de magnitude, é empregar arquiteturas emergentes de programação GPGPU (*General Purpose computing on Graphics Processing Unit*), como a plataforma de programação CUDA [Owens J. e Purcell, 2007, Nickolls et al., 2008].

1.2 Definição do Problema e Motivação

As abordagens propostas na literatura para a busca e a mineração sobre dados complexos diferem, principalmente, na maneira como o problema é modelado. Nas áreas de banco de dados e recuperação da informação, destacam-se duas linhas de pesquisa: os métodos de busca exata e os métodos de busca aproximada.

Na linha de pesquisa dos métodos de busca exata, os Métodos de Acesso Espaciais (MAEs) e os Métodos de Acesso Métricos (MAMs) são os mais conhecidos e usados em muitas aplicações. Uma classe importante de Métodos de Acesso é aquela em que os métodos são baseados em estruturas hierárquicas em árvore, que adotam um processo recursivo para a indexação dos dados. A maioria dessas abordagens de busca são baseadas em algoritmos determinísticos, já estudados há muitos anos.

No entanto, a busca por similaridade em altas dimensões pela abordagem hierárquica é custosa, pois acima de certa dimensão, esses métodos degradam sua performance e, no pior caso, todas as regiões são acessadas e a busca torna-se sequencial [Böhm et al., 2001]. A razão disso é que a organização hierárquica dos dados torna-se mais difícil com o aumento da dimensionalidade, pois os valores de distância, para a maioria de pares de objetos estão muito próximos e, como consequência, o fator de poda de regiões nas consultas diminui.

Os métodos de acesso, tanto os métricos quanto os espaciais são sensíveis a este problema. Do ponto de vista prático, a indexação hierárquica só é eficaz quando as buscas acessam menos de 10% das

regiões [Blott e Weber, 2008], caso contrário, a busca sequencial e métodos baseados em *hash* tornam-se soluções viáveis ao se considerar o aumento da dimensionalidade.

Assim, numa segunda linha de pesquisa, voltada para a busca aproximada e algoritmos probabilísticos, pesquisadores têm tentando reduzir os efeitos da "maldição da alta dimensionalidade" relaxando a precisão da consulta, a fim de reduzir o custo computacional da mesma. Isto é válido em aplicações em que a resposta exata não é um requisito fundamental. Além disso, como a própria definição intuitiva de similaridade já envolve uma aproximação para a resposta de uma consulta, uma segunda aproximação pode ser aceitável em diversos contextos [Chávez et al., 2001].

Nessa linha, a técnica *Locality Sensitive Hashing* (LSH) [Datar et al., 2004, Andoni e Indyk, 2008] foi proposta para realizar a busca aproximada por similaridade em altas dimensões com custo sublinear. A ideia básica é que se dois objetos são similares, eles são mapeados no mesmo *bucket* com alta probabilidade. Esse mapeamento é basedo em técnicas de projeção aleátoria em subespaços.

Na área de mineração de dados, o LSH tem sido utilizado com êxito para melhorar diversos algoritmos, como a identificação de agrupamentos [Hisashi Koga, 2004, Hisashi Koga, 2007], reconhecimento de padrões [Wang et al., 2010] e a descoberta de *motifs* [Lin et al., 2002]. A razão disso é que, se o LSH é configurado adequadamente, ele é um dos poucos métodos que asseguram um custo de busca sublinear [Datar et al., 2004, Andoni e Indyk, 2008]. Entretanto, alguns inconvenientes do LSH ainda precisam ser resolvidos. Em particular, identificou-se que muitas das implementações atuais do LSH ou incorrem em um alto custo de memória ou abandonam a garantia teórica de custo sublinear, para garantir qualidade nos resultados [Tao et al., 2010].

Na área de mineração de dados, diversos algoritmos de mineração de dados baseados em buscas ao vizinho mais próximo foram propostos [Liu et al., 2004, Liu et al., 2006] e muitos deles fornecem resultados exatos. Entretanto, apesar das recentes pesquisas em algoritmos exatos, os algoritmos aproximados podem ser a melhor opção em muitos domínios de aplicação, devido a sua eficiência em tempo e espaço. Nesses domínios, o *trade-off* entre o tempo de execução e a precisão da solução claramente se inclina para a primeira. Esse fato é importante para o embasamento da proposta deste trabalho.

Considerando que ainda muitas tarefas de mineração de dados utilizam a busca kNN como procedimento principal, melhorar o desempenho do processo de busca kNN é um aspecto importante. No entanto, mesmo considerando soluções aproximadas como uma abordagem para acelerar consultas por similaridade, muitas soluções simples e gerais de problemas que exploram a busca kNN ainda são consideradas impraticáveis, devido a seu alto custo computacional. Uma opção para desenvolver soluções de alto desempenho para tarefas que demandem grandes recursos de computação é a utilização de esquemas de programação paralelas.

Como uma alternativa para soluções de alto desempenho, a capacidade computacional das GPUs tem sido explorada para melhorar muitas das tarefas de recuperação de informação e de mineração de dados. Devido ao crescente poder das unidades de processamento gráfico, soluções baseadas em GPUs podem ser adequadas para melhorar o desempenho dos algoritmos. Embora as GPUs sejam normalmente utilizadas para acelerar processos gráficos altamente exigentes, após a introdução do conceito de computação GPGPU (*General Purpose computing on Graphics Processing Unit*), as GPUs passaram a fornecer grandes recursos de computação de propósito geral para problemas altamente paralelizáveis. Entre as plataformas de programação GPGPU, destacam-se: CUDA [Nickolls et al., 2008] e OpenCL

[Stone et al., 2010]. A arquitetura CUDA foi escolhida para este trabalho, porque no momento da escrita do texto era a plataforma dominante para computação GPGPU [Nickolls et al., 2008].

Em geral, nas aplicações em que o desempenho é um fator essencial, as técnicas recentes propiciam tanto o relaxamento da precisão dos resultados como a utilização das técnicas de programação GPGPU, a fim de reduzirem o custo computacional dos algoritmos. Desse modo, o potencial da abordagem de busca aproximada associada à computação GPGPU para o desenvolvimento de novos algoritmos de mineração de dados, especificamente para descoberta de *motifs*, são as principais motivações deste trabalho de mestrado.

1.3 Objetivos

Este trabalho tem como objetivo investigar e desenvolver soluções baseadas em busca por similaridade aproximada e a computação GPGPU, como forma de reduzir o custo computacional de tarefas de mineração de dados que utilizam o kNN como procedimento mais importante, especificamente a descoberta de *motifs*. Sendo assim, foram desenvolvidas soluções e algoritmos com o objetivo de criar métodos que permitam responder as seguintes questões:

1. Existem soluções aproximadas de busca por similaridade com uma análise teórica sólida de custo de busca sublinear, sendo esta adequada para uma eficiente implementação na plataforma de programação CUDA?

2. Como soluções de busca paralela kNN podem ser usadas para acelerar tarefas de mineração de dados?

Portanto, este trabalho explora técnicas aproximadas de busca ao vizinho mais próximo, a fim de encontrar a solução mais adequada para resolver este tipo de busca em domínios de dados complexos e para, especificamente, trabalhar com dados multidimensionais, considerando sempre o aprimoramento de soluções simples para sua implementação eficiente em CUDA.

Uma vez determinada esta solução, a abordagem de busca paralela é estendida para melhorar o desempenho das tarefas de mineração em séries temporais, e, de modo específico, para desenvolver soluções de alto desempenho para descoberta de *motifs*.

1.4 Principais Contribuições

A principal contribuição deste trabalho é o desenvolvimento de algoritmos de descoberta de *motifs* que utilizam a abordagem de busca por similaridade aproximada e computação GPGPU. Várias das soluções propostas na literatura para a identificação de *motifs* são consideradas impraticáveis em um contexto real, devido ao seu alto custo computacional, mas graças ao uso de abordagens aproximadas associadas à computação GPGPU, foi possível desenvolver algoritmos de alto desempenho para grandes bases de dados, apresentando, dessa forma, um bom equilíbrio entre a velocidade e precisão.

Assim, as soluções de busca paralela kNN baseadas no esquema de indexação LSH, foram implementadas em CUDA e estendidas para melhorar o desempenho de tarefas de mineração em séries temporais, em particular, para desenvolver soluções de alto desempenho de descoberta de *motifs*.

No Capítulo 5 é apresentado o método de indexação de dados multidimensionais em GPU, chamado de CUDA-LSH. Ele foi desenvolvido para fornecer buscas de alto desempenho para consultas por similaridade aproximada em conjuntos de dados multidimensionais, especificamente a busca paralela AllkNN, implementada com o algoritmo *CUDA-AllkNN()* . Além disso, desenvolveu-se soluções de alto desempenho para a identificação dos *TopK-Motifs* , implementada com o algoritmo *CUDA-TopKMotifs()* e *CUDA-RandomProjection()*.

Outra contribuição é a avaliação empírica da proposta com métodos recentes para busca por similaridade e descoberta de *motifs*, assim como a analise tanto quantitativa como qualitativa das técnicas de busca kNN e descoberta de *motifs* (Capítulo 6).

Finalmente, a análise sobre o estado da arte em soluções aproximadas de busca por similaridade e descoberta de *motifs* é mais uma contribuição, pois incide sobre as abordagens de interesse prático para banco de dados e mineração de dados de soluções aproximadas para trabalhar com dados em altas dimensões (Capítulo 3 e 4).

1.5 Organização do Trabalho

Este trabalho está organizado em sete capítulos, incluindo a presente introdução, e tem a seguinte estrutura:

- No **Capítulo 2** irá se descrever os principais conceitos relacionados a consultas por similaridade em espaços multidimensionais e espaços métricos.

- No **Capítulo 3** serão abordados os principais conceitos relacionados à mineração de dados. Destaca-se o processo de descoberta de *motifs* e consultas por similaridade em séries temporais, sendo os mesmos de interesse neste trabalho.

- No **Capítulo 4** serão introduzidos o modo de programação GPGPU, a arquitetura e as dificuldades tanto das GPUs tradicionais quanto da nova arquitetura para programação em GPUs CUDA.

- No **Capítulo 5** serão apresentadas as implementações da solução desenvolvida do LSH em CUDA, assim como os algoritmos de descoberta de *motifs* baseados nesta solução de busca de alto desempenho.

- No **Capítulo 6** serão tratados os estudos experimentais realizados com conjuntos de dados sintéticos e reais para busca por similaridade e descoberta de *motifs*.

- No **Capítulo 7** serão feitas as considerações finais, incluindo principais contribuições e propostas de trabalhos futuros.

CAPÍTULO 2

Consulta por similaridade sobre dados complexos

2.1 Considerações Iniciais

Na área de banco de dados, as técnicas de indexação têm como objetivo auxiliar o armazenamento e a recuperação eficiente de dados. Muitas dessas técnicas são aplicadas a tarefas de recuperação de informação. Em geral, essa organização é realizada por meio de uma estrutura de indexação, também referenciada como Método de Indexação (MI) ou Método de Acesso (MA). Aplicações de bases de dados empregam métodos de acesso como os Métodos de Acesso Espaciais (MAEs) e os Métodos de Acesso Métricos (MAMs) devido à sua capacidade de construir a estrutura de dados para gerenciar e organizar grandes conjuntos de dados de maneira eficiente.

No entanto, devido à "maldição da alta dimensionalidade", o tempo necessário para realizar buscas por similaridade, como o kNN, pode crescer exponencialmente com a dimensionalidade dos dados. Embora nenhum método seja conhecido por um bom desempenho em todas as instâncias do problema, existem muitos algoritmos para acelerar as consultas, normalmente por meio de uma aproximação da resposta correta. Assim, define-se dois modelos de busca: os métodos de busca exata e os métodos de busca aproximada.

Na linha de pesquisa dos métodos de busca exata, os Métodos de Acesso Espaciais (MAEs), como Kd-Tree [Bentley, 1979], R-Tree [Guttman, 1984], R*-Tree [Beckmann et al., 1990], R+-Tree [Sellis et al., 1987] e X-Tree [Berchtold et al., 1996], descrevem os dados de entrada como vetores multidimensionais. Esses métodos foram concebidos para apoiar as operações de busca envolvendo pontos e objetos geométricos [Gaede e Gunther, 1998].

Ainda nessa linha, Faloutsos [Faloutsos et al., 1994] propõem o *framework* GEMINI *(GEneric Multimedia INdexIng)*, que utiliza os MAEs junto com métodos redução de dimensionalidade para lidar com dados em alta dimensionalidade. A ideia geral desse *framework* consiste em reduzir o custo de busca com a utilização de um esquema para a identificação de falsos alarmes, visando descartar a maioria dos

objetos que não qualificam como candidatos.

Na mesma direção dos métodos de busca exata, os Métodos de Acesso Métricos (MAMs) fornecem operações de busca por similaridade, exata, em espaços métricos. As estruturas de indexação dos MAMs organizam os dados usando um critério de similaridade, visando uma resposta eficiente nas consultas. Na literatura são encontrados MAMs baseados em árvores, como VP-Tree [Yianilos, 1993], SAT [Navarro, 2002], M-Tree [Ciaccia et al., 1997], Slim-Tree [Traina C. et al., 2002], DBM-Tree [Vieira et al., 2004], DF-Tree [Traina et al., 2002] e PM-Tree [Skopal et al., 2005], e as técnicas baseadas em grafos, como t-Spanners [Navarro et al., 2007] e HRG [Ocsa et al., 2007]. Estudos extensos sobre MAMs podem ser encontrados em [Chávez et al., 2001, Hjaltason e Samet, 2003, Clarkson, 2006].

Os MAMs utilizam intensamente a propriedade da desigualdade triangular para reduzir o número de cálculos de distância (ver Seção 2.2.1). No entanto, embora muitos MAMs tenham sido propostos para acelerar a busca por similaridade, alguns ainda sofrem com o problema de sobreposição. Além disso, alguns trabalhos de pesquisa discutem a ideia de que indexação hierárquica de dados em altas dimensões pode deteriorar as consultas, mesmo em comparação com a busca sequencial [Böhm et al., 2001, Blott e Weber, 2008].

Numa outra abordagem, uma técnica promissora chamada *Locality Sensitive Hashing* (LSH) [Datar et al., 2004] foi proposta para realizar consulta por similaridade aproximada em dados em altas dimensões, de maneira eficiente. O LSH, discutido em detalhes na Seção 2.5.2, é baseado na ideia de que a proximidade entre dois objetos é geralmente preservada por uma operação de projeção randômica. Em outras palavras, se dois objetos são próximos em seu espaço original, então eles permanecem próximos depois de uma operação de projeção.

Na Figura 2.1 são ilustrados os modelos de busca por similaridade para as abordagens do LSH e dos MAMs. O modelo unificado para busca por similaridade em espaços métricos apresentado em [Chávez et al., 2001] é ilustrado na Figura 2.1 (a). Um índice métrico organiza o conjunto de dados em regiões definidas pela função de distância, de modo que cada região inclui objetos que são suficientemente próximos uns dos outros. No momento da consulta, a desigualdade triangular é usada para eliminar as regiões que não se sobrepõem à região de consulta. Assim, apenas os objetos em regiões candidatas (regiões cinza na figura) são posteriormente tratados, a fim de testar a condição da consulta.

Um modelo de pesquisa similar para LSH é ilustrado na Figura 2.1 (b), onde cada um dos três subíndices é definido por um conjunto de funções *hash* (H_1, H_2, H_3), gerando três tipos de particionamento do espaço de busca. Assim, cada partição está associada a uma tabela *hash* e seu correspondente conjunto de funções *hash*. Cada conjunto de funções *hash* é utilizado para organizar o conjunto de dados em regiões, de modo que com certa probabilidade, os objetos na mesma região são considerados o suficientemente próximos. Em tempo de consulta, o objeto de consulta q é projetado (usando cada um dos três conjuntos de funções *hash*, um por cada partição) em regiões onde a probabilidade de encontrar objetos próximos é muito alta (regiões cinza na figura). Finalmente, já que são considerados vários subíndices, as regiões candidatas são analisadas a fim de retornar só os objetos que satisfazem a condição da consulta e que não tenham sido retornados antes.

Na Seção 2.2 são apresentadas as definições elaboradas em [Chávez et al., 2001], cuja abordagem estabelece os conceitos básicos de busca em domínios multidimensionais e métricos. Na Seção 2.3 são apresentados os tipos de consulta por similaridade mais comuns suportados pelos métodos de busca exata e aproximada em domínios multidimensionais e métricos. Na Seção 2.4 é discutido como a busca por

(a) (b)

Figura 2.1: Unificação do modelo de consulta por similaridade. (a) Os Métodos de Acesso Métricos (MAMs). (b) *Locality Sensitive Hashing* (LSH).

similaridade é afetada pela dimensionalidade dos dados. Na Seção 2.5 são descritos em profundidade alguns dos principais métodos aproximados encontrados na literatura.

2.2 Domínio dos Dados

A concepção do método de busca deve considerar a natureza dos dados, a fim de explorar certas propriedades do espaço em que estão inseridos.

A grande maioria dos métodos de busca supõe que os dados pertencem a um espaço n-dimensional. Esses espaços são isoformas a ^{n}, o que permite o uso de propriedades geométricas na solução. Por outro lado, alguns métodos tentam fazer exigências menos rigorosas, como o caso dos métodos baseados em espaços métricos, que só usam as distâncias entre os pontos, e nenhuma outra propriedade geométrica.

2.2.1 Espaços Métricos

Um espaço métrico é definido como $=< S, d >$, onde S é um universo de elementos e d é a função de distância (ou métrica), definida sobre os elementos em S, que mede a dissimilaridade entre os objetos e satisfaz as seguintes condições, $\forall x, y, z \in S$:

(1) $d(x, y) \geq 0$	positividade
(2) $d(x, y) = 0 \leftrightarrow x = y$	reflexividade
(3) $d(x, y) = d(y, x)$	simetria
(4) $d(x, y) \leq d(x, z) + d(y, z)$	desigualdade triangular

A desigualdade triangular é considerada uma das propriedades mais importantes, pois é utilizada para estabelecer os limites do valor da distância entre dois objetos sem a necessidade do cálculo real da distância, acelerando assim os algoritmos de consulta por similaridade. Mais especificamente, dados os valores de distância $d(x, z)$ e $d(y, z)$, os limites para o valor (desconhecido) de $d(x, y)$ são $|d(x, z) - d(y, z)| \leq d(x, y) \leq d(x, z) + d(y, z)$.

Nem todas as propriedades (1)-(4) são necessárias para todos os métodos. Por exemplo, pode-

se substituir (2) por uma propriedade mais fraca $\forall x \in S, d(x,x) = 0$, tornando o espaço pseudo-métrico. Os métodos que não obedecem às propriedades (3), (4) ou ambas são tipicamente chamados não métricos. Nesses casos, costuma-se usar o nome mais geral da **dessemelhança** para a função d, em vez de **distância**, que é tradicionalmente usado para espaços métricos.

2.2.2 Espaços Multidimensionais

Se os objetos do domínio S correspondem a vetores de valores numéricos então o espaço é chamado Espaço Multidimensional ou Espaço Vetorial com Dimensão Finita. Os objetos de um espaço multidimensional de dimensão n (ou n-dimensional) são representados por n coordenadas de valores reais $x_1, ..., x_n$.

As funções de distância métrica mais comuns para mensurar a similaridade entre elementos em um Espaço Multidimensional são as da família L_p, ou Minkowski, definidas por:

$$L_p((x_1, ..., x_n), (y_1, ..., y_n)) = (\sum_{i=1}^{n} |x_i - y_i|^p)^{1/p} \tag{2.1}$$

- A distância L_1 é também conhecida como distância **Manhattan**, e corresponde à simples soma das diferenças absolutas dos componentes.
- A distância L_2 é também conhecida como distância **Euclidiana**, e corresponde à ideia usual de distância em espaços $2D$ ou $3D$.
- A distância L_∞ é também conhecida como **Chebychev**, e corresponde à diferença máxima absoluta entre os componentes, definida por:

$$L_\infty((x_1, ..., x_n), (y_1, ..., y_n)) = max_{i=1}^{n} |x_i - y_i| \tag{2.2}$$

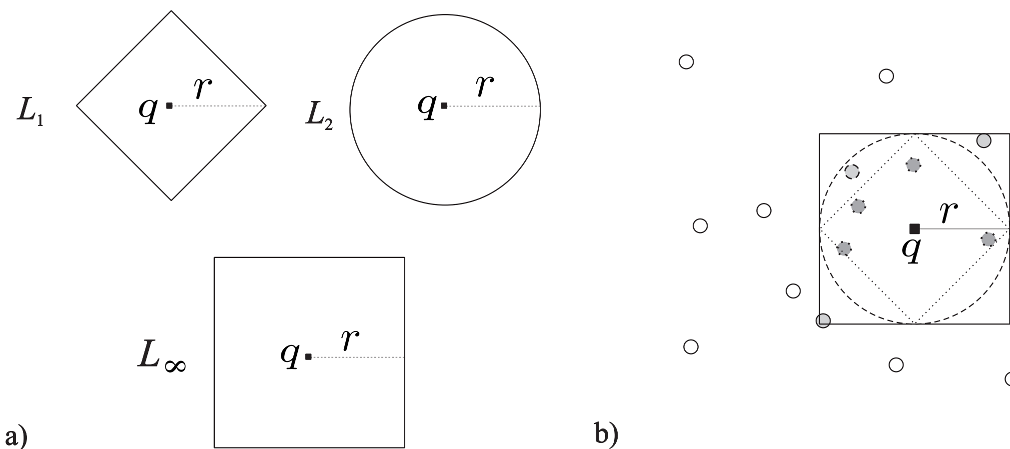


Figura 2.2: (a) Formas geométricas que ilustram a delimitação da região de busca de acordo com a métrica L_p utilizada. (b) Exemplo de consulta por abrangência para diferentes métricas da família L_p.

A Figura 2.2(a) ilustra, para diferentes distâncias da família L_p, o conjunto de pontos que estão à mesma distância r, a partir de um centro q, num espaço bidimensional.

A Figura 2.2(b) representa um conjunto de objetos num espaço bidimensional, um objeto de consulta q, o raio de consulta r, os conjuntos de resposta e três métricas utilizadas: L_1, L_2 e L_∞.

2.3 Consultas por Similaridade

As consultas por similaridade utilizam critérios de busca baseados na semelhança entre o objeto de busca e os objetos armazenados na base de dados. A similaridade (ou a dissimilaridade) entre dois objetos é medida por meio de uma função de distância, que deve ser definida de acordo com o domínio dos dados. Os tipos de consultas por similaridade mais comuns são as consultas por abrangência (*range queries*) e as consultas pelos k-vizinhos mais próximos (*k-nearest neighbor queries*).

Em aplicações de recuperação de informação por similaridade, usualmente define-se um universo de objetos S e uma função $d : S \times S \rightarrow {}^{+}$ que mede a distância entre dois objetos de S. Assim, dado um objeto de consulta $q \in S$, objetos semelhantes a q podem ser recuperados de acordo com os principais tipos de consultas por similaridade, definidos a seguir.

Consulta por Abrangência - $Rq(q, r)$ Consulta que visa recuperar objetos semelhantes a q que se encontram dentro do raio de consulta r. Ou, mais formalmente:

$$Rq(q, r) = \{u \in S | d(u, q) \leq r\} \tag{2.3}$$

Na Figura 2.3 é apresentada uma consulta por abrangência num espaço bidimensional com a métrica Euclidiana L_2. Os elementos contidos pelo raio de cobertura r compõem a resposta. Vale lembrar que não é necessário que o elemento de consulta pertença ao conjunto de dados de pesquisa, devendo pertencer ao mesmo domínio dos dados.

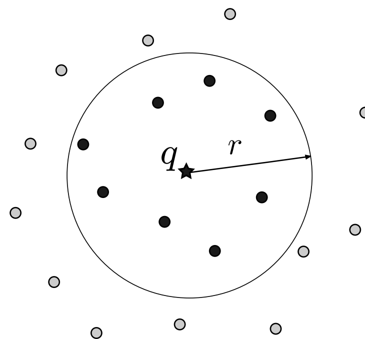


Figura 2.3: Exemplo de consulta por abrangência.

Consulta aos k-Vizinhos mais Próximos - $kNN(q, k)$ Consulta que visa recuperar os k objetos mais próximos ao objeto de busca q. Os k vizinhos mais próximos definem um conjunto $C = \{s_1, s_2, ..., s_k\}$ em que:

$$\forall s_i \in C, \forall x_j \in S - C, d(q, s_i) \leq d(q, x_j) \tag{2.4}$$

A Figura 2.4 exemplifica uma consulta kNN num espaço bidimensional com a métrica Euclidiana L_2. Na figura, a consulta tem como entrada o elemento de consulta q e o valor de k igual 3. Apenas os

elementos ligados por traços (os mais próximos) compõem a resposta, considerando que no exemplo o elemento q não pertenece ao conjunto de dados.

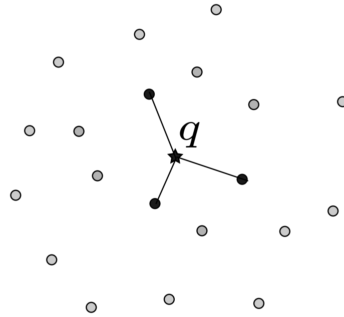


Figura 2.4: Exemplo de consulta por vizinhos mais próximos.

Consulta a todos os k-Vizinhos mais Próximos - $All - kNN(Q, k)$ Consulta que visa recuperar todos os k objetos mais próximos a todos os objetos de busca $q \in Q$, em que Q é o conjunto de dados original, ou seja $Q = S$.

Tal como a busca aos k-vizinhos mais próximos (kNN) pode ser restringida só para uma busca ao vizinho mais próximo (NN), este tipo de consulta massiva (AllkNN) também pode ser reduzida a um tipo de consulta chamada $All - NN(Q)$. Nesta consulta, todos os vizinhos mais próximos aos elementos de Q são procurados, e apenas aqueles cuja distância seja diferente de zero são retornados, ou seja, não se podem retornar como conjunto de resposta elementos iguais aos objetos de consulta. A Figura 2.5 ilustra a noção do problema de busca AllkNN para um conjunto de dados bidimensional.

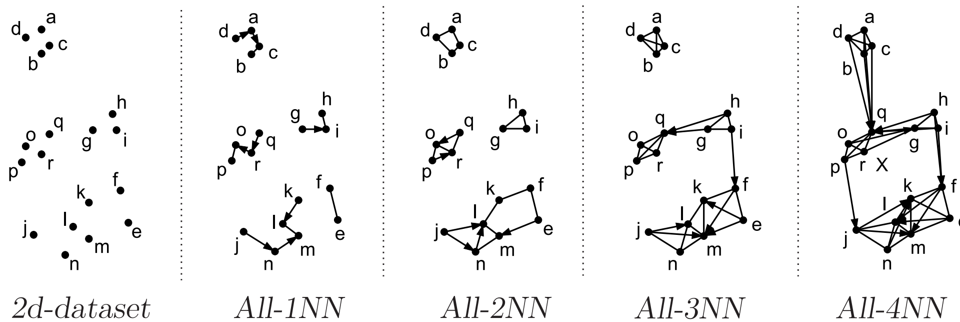


Figura 2.5: Exemplo de consultas AllkNN para um conjunto de dados bidimensional.

Para as consultas por similaridade, de interesse neste trabalho, uma solução óbvia é a busca sequencial, em que cada elemento da base é avaliado em relação ao objeto de consulta e assim os objetos dentro do raio r ou os k mais similares são retornados dependendo do tipo da consulta. Entretanto, essa solução de força bruta apenas é aceitável para pequenas bases de dados, sendo inviável para problemas de larga escala, como por exemplo a busca de imagens por conteúdo na Web. Além disso, como discutido na Seção 2.4, em algumas aplicações a pesquisa kNN como método de busca exata é tão custosa que, muitas vezes, as soluções aproximadas são aceitáveis. Nesses casos, a utilização das definições a seguir serão usadas.

Consulta por Abrangência Aproximada - $(1+\varepsilon) - Rq(q,r)$ Consulta que visa recuperar os objetos semelhantes a q que se encontram dentro do raio de consulta $(1+\varepsilon) \times r$. Ou, mais formalmente:

$$(1+\varepsilon) - Rq(q,r) = \{u \in S | d(u,q) \leq (1+\varepsilon) \times r\} \tag{2.5}$$

A Figura 2.6 exemplifica uma consulta deste tipo num espaço bidimensional com a métrica Euclidiana L_2. Os elementos contidos pelo raio de cobertura $(1+\varepsilon) \times r$ compõem a resposta aproximada.

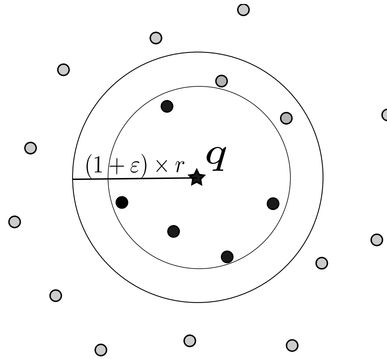


Figura 2.6: Exemplo de consulta por abrangência aproximada.

Consulta Aproximada aos k-Vizinhos mais Próximos - $(1+\varepsilon) - kNN(q,k)$ Seja r a distância entre o objeto de consulta q e o elemento mais distante entre os k verdadeiros vizinhos mais próximos, isto é,

$$r = max\{s_j \in S | d(q, s_j)\} \tag{2.6}$$

A busca $(1+\varepsilon) - kNN$ para os k elementos mais similares a q consiste em encontrar um conjunto $C' = \{s'_1, s'_2, ..., s'_k\}$ em que,

$$\forall s'_i \in C', d(q, s'_i) \leq (1+\varepsilon) \times r \tag{2.7}$$

O fator $(1+\varepsilon)$ é usualmente chamado **fator de aproximação**, e indica que o conjunto de solução C' está dentro de um **erro relativo** ε da resposta exata. É possível dar uma interpretação geométrica para o erro relativo: se a esfera mínima delimitadora centrada em q e contendo C' tem um raio r, então a esfera mínima delimitadora centrada em q e contendo C' tem um raio de $(1+\varepsilon) \times r$, como ilustrado na Figura 2.7.

Consulta Aproximada a todos os k-Vizinhos mais Próximos - $(1+\varepsilon) - AllkNN(Q,k)$ Este tipo de consulta aproximada é baseado na definição da consulta exata $All - kNN(Q,k)$ e a consulta aproximada $(1+\varepsilon) - kNN(q,k)$. Assim, neste problema, utilizando a definição aproximada da busca kNN, procura-se os k-vizinhos mais próximos a cada objeto $q \in Q$, em que Q é o conjunto de dados original, ou seja $Q = S$.

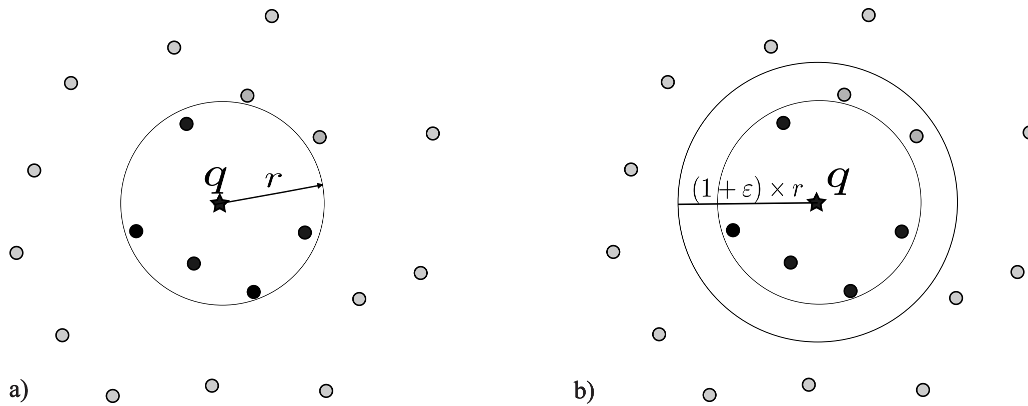


Figura 2.7: Interpretação geométrica de uma busca aproximada 5-NN. (a) busca exata. (b) busca aproximada. O objeto de consulta é o ponto central q, os pontos pretos compõem o conjunto solução.

2.4 A Maldição da Alta Dimensionalidade

A eficiência dos métodos de busca por similaridade depende muito da dimensionalidade dos elementos. Embora o tempo de busca possa atingir um custo logarítmico com relação ao tamanho do banco de dados, ele vai crescer exponencialmente com a dimensionalidade dos elementos.

Para dimensões moderadas, a maioria dos métodos executam as consultas de maneira suficientemente eficiente para permitir uma solução exata ou quase exata em tempo razoável. Para dimensões altas, torna-se viável o uso de métodos aproximados, o que implica em um *trade-off* entre a precisão e a eficiência. Para dimensões muito altas esse *trade-off* vai ficando progressivamente mais relevante, resultando em penalidades maiores em termos de precisão, a fim de obter uma eficiência aceitável.

O problema da "maldição da alta dimensionalidade" foi estudado originalmente por Bellman [Bellman, 1961], observando que partições do espaço de soluções em problemas de otimização são ineficientes em problemas com altas dimensões. Os efeitos da "maldição" em Métodos de Acesso Espaciais (MAEs) e Métodos de Acesso Métricos (MAMs) são discutidos em [Böhm et al., 2001, Blott e Weber, 2008, Volnyansky e Pestov, 2009].

2.5 Abordagens para a Solução Aproximada

A literatura sobre os métodos de busca por similaridade é bastante vasta, com dezenas de métodos e centenas de variantes. Apesar de haver uma evidente produção científica nessa área, a abundância torna inviável explorar exaustivamente o estado da arte neste trabalho. Assim, o objetivo nesta seção não é descrever todos os métodos de indexação, mas sim os mais relevantes para esta pesquisa, pois como será visto no Capítulo 5 a proposta do trabalho centra-se em soluções aproximadas de busca por similaridade baseadas em projeções em subespaços.

Com esse foco, na Seção 2.5.1 descreve-se os métodos de indexação baseados nas *space-filling curves*, na Seção 2.5.2 descreve-se o *Locality Sensitive Hashing* (LSH) e na Seção 2.5.3 descreve-se o HashFile.

2.5.1 Space-Filling Curves

As chamadas *space-filling curves* são associações de endereços (valor unidimensional) a pontos no espaço multidimensional, que fazem uso de funções de mapeamento. A existência dessas técnicas de mapeamento contínuo de unidades de intervalo [0:1] e qualquer hipercubo $[0;1]^n$ implica a existência de curvas contínuas que cobrem completamente qualquer hipercubo $n-$dimensional. A Figura 2.8 ilustra alguns exemplos de *2D space-filling curves*.

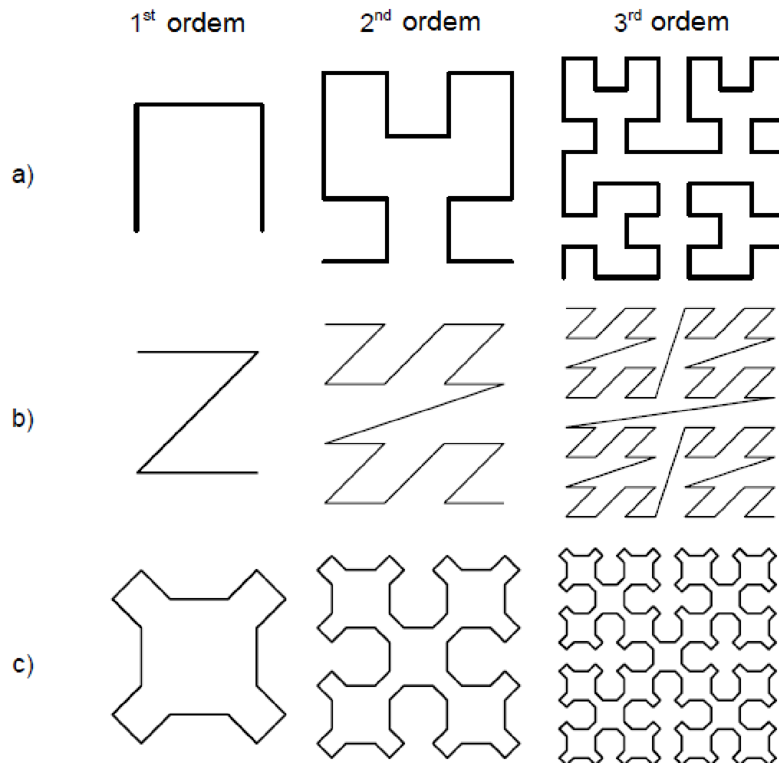


Figura 2.8: Alguns exemplos de *2D space-filling curves*. a) Hilbert; b) Lebesgue ou "Z-order"; c) Sierpinski. As três iterações de cada curva são apresentadas [Valle, 2008].

A maioria das *space-filling curves* são construídas através de um processo recursivo, em que cada espaço é dividido progressivamente em células menores, que são depois atravessadas pela curva. Nesse caso, o fim da curva é o número de iterações que foram realizadas na série infinita da mesma. O espaço é preenchido pela curva em que a série converge. Uma curva de ordem finita é dita ser uma aproximação do chamado *space-filling curve* [Sagan, 1994].

O uso das teorias das *space-filling curves* para realizar busca kNN em espaços multidimensionais não é nova. Faloutsos [Faloutsos, 1986] foi um dos primeiros autores a utilizar conceitos das *space-filling curves*. Faloutsos foi o primeiro a sugerir que outras curvas poderiam ter um melhor desempenho, primeiro propondo a curva *gray-code*, e depois, a curva de Hilbert [Faloutsos, 1986, Faloutsos e Roseman, 1989].

Todos esses métodos são conceitualmente muito simples. Eles mapeiam objetos em altas dimensões na curva, resultando em um valor unidimensional, chamado de chave estendida. Essa chave representa sua posição relativa no comprimento da curva e é usada para executar consultas por similaridade usando índices unidimensionais. A hipótese é que os objetos que estão próximos uns dos outros na curva sempre

correspondem aos objetos que estão próximos no espaço original.

No entanto, o inverso da hipótese não é verdadeiro, no sentido de que objetos próximos no espaço nem sempre estão perto da curva. Isso ocorre por causa dos efeitos de fronteira, que tendem a colocar objetos distantes em certas regiões próximas na curva. A Figura 2.9 ilustra o problema com o efeito de fronteira nas *space-filling curves*.

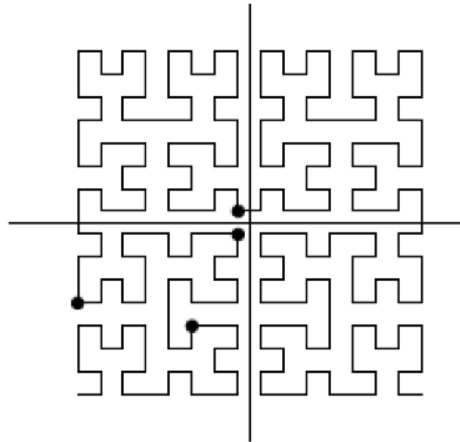


Figura 2.9: O problema com o efeito de fronteira nas *space-filling curves*. Os pontos no centro do espaço estão mais afastados na curva do que os pontos do quadrante inferior esquerdo [Valle, 2008].

Este efeito é agravado com o aumento da dimensionalidade. Para resolver esse problema, diferentes autores propõem soluções distintas, geralmente baseadas no uso de várias curvas [Shepherd et al., 1999, Liao et al., 2001].

2.5.2 *Locality Sensitive Hashing*

Alguns trabalhos em busca aproximada [Gionis et al., 1999, Datar et al., 2004, Andoni e Indyk, 2008] exploram a ideia de mapear os objetos de um conjunto de dados e agrupá-los em *buckets* com o objetivo de executar consulta por similaridade aproximada dentro dos *buckets* associados ao elemento de consulta. Em particular, o *Locality Sensitive Hashing* (LSH) foi concebido para resolver eficientemente consultas por abrangência aproximada $((1+\epsilon) - Rq(q, r))$. Como definido na Seção 2.3, esse tipo de consulta visa recuperar os objetos que se encontram dentro do raio de consulta $(1+\epsilon) \times r$. A ideia principal é que, se dois objetos são próximos no espaço original, esses dois objetos tendem a permanecer próximos depois de uma operação de projeção escalar randômica. Assim, dada uma função *hash* $h(x)$ que faz o mapeamento de objetos n-dimensionais (x) a um valor unidimensional, a função é sensível à localidade se a possibilidade de mapeamento de dois objetos x_1, x_2 ao mesmo valor cresce à medida que a distância $d(x_1, x_2)$ diminui. Formalmente:

Definição Dados um valor de distância r, um fator de aproximação $(1+\epsilon)$, as probabilidades P_1 e P_2, tal que $P_1 > P_2$, a função *hash* $h()$ é sensível à localidade se satisfaz as condições a seguir:

- $Se\ d(x_1, x_2) \leq r \Rightarrow Pr[h(x_1) = h(x_2)] \geq P_1.$

  Isto quer dizer que para dois objetos x_1 e x_2 in R^n que estão o suficientemente próximos, há uma alta probabilidade P_1 que eles caiam no mesmo *bucket*.

- *Se* $d(x_1, x_2) > (1 + \epsilon) \times r \Rightarrow Pr[h(x_1) = h(x_2)] \leq P_2$.

  Isto quer dizer que para dois objetos x_1 e x_2 in R^n que estão distantes, há uma probabilidade baixa $P_2 < P_1$ que eles caiam no mesmo *bucket*.

O esquema proposto em [Datar et al., 2004] foi projetado para trabalhar com distribuições p-estáveis da seguinte maneira: calcular o produto escalar $\vec{a} \cdot \vec{x}$ para atribuir um valor *hash* para cada vetor x. A função *hash* precisa de valores randômicos $\vec{a}$ e b, em que $\vec{a}$ é um vetor n-dimensional com valores independentemente escolhidos a partir de distribuições p-estáveis (Cauchy ou Gaussiana) e b é um número real uniformemente escolhido dentro de um intervalo $[0, \omega]$. Portanto a função *hash* $h(x)$ é dada por:

$$h(x) = \lfloor \frac{\vec{a} \cdot \vec{x} + b}{\omega} \rfloor \quad (2.8)$$

A Equação 2.8 tem uma interpretação geométrica simples. Considere-se a Figura 2.10: $\vec{p_1}$ e $\vec{p_2}$ são dois vetores em $\mathbf{R}^2$, $\vec{a}$ é o vetor normal unitário e b é um número randômico real. A inclinação da reta passando pela origem coincide com a direção do $\vec{a}$. Assim, $\vec{p_1}$ é projetado sobre a linha $\vec{a}$ calculando o produto escalar $\vec{a} \cdot \vec{p_1}$. Essa projeção é quantizada em intervalos de tamanho fixo ω, definindo o ponto A. O mesmo procedimento é repetido para $\vec{p_2}$, definindo o ponto B.

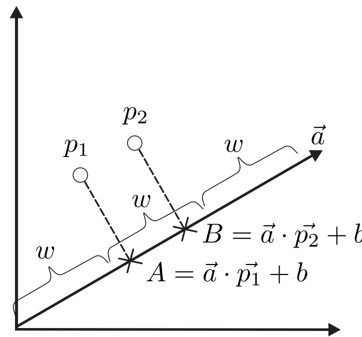


Figura 2.10: Interpretação geométrica da projeção dos vetores $\vec{p_1}$ e $\vec{p_2}$ em LSH.

Para ampliar a diferença entre as probabilidades P_1 e P_2 pode-se usar m funções *hash* diferentes. Isto aumenta a proporção das probabilidades dado que $(P_1/P_2)^m > P_1/P_2$, assegurando assim que se dois objetos estão muito distantes, a probabilidade de caírem no mesmo *bucket* será baixa. No entanto, embora seja importante evitar que objetos distantes caiam no mesmo *bucket* para preservar a proximidade espacial, isso não é uma condição suficiente. É igualmente importante assegurar que objetos próximos no espaço original apareçam no mesmo *bucket* com alta probabilidade. Entretanto, como isso não pode ser feito usando só uma estrutura *hash*, LSH resolve esse problema considerando L projeções independentes, ou seja, a construção de L tabelas *hash* (L subíndices) [Slaney e Casey, 2008, Tao et al., 2010].

Assim, usando coleções diferentes de m funções *hash* $H = \{h_1, h_2, ..., h_m\}$, sendo uma coleção para cada subíndice, cada objeto x de um conjunto de dados é mapeado em *buckets*, efetivamente particionando o conjunto de dados em vários grupos pequenos (os *buckets*) para cada um dos L subíndices. A estrutura de dados será então formada por L subíndices. LSH divide o espaço de busca empregando m funções *hash* escolhidas aleatoriamente de uma distribuição Gaussiana (ou Cauchy). O número de funções *hash* m determina o quão esparso ou denso será o espaço de busca. Ao aumentar

o valor de m, os objetos tendem a ser distribuídos em *buckets* de maneira bastante uniforme, reduzindo a precisão de consultas já que é mais provável que objetos similares caiam em *buckets* diferentes. Para atenuar esse efeito negativo, são necessárias muitas tabelas *hash*. Por outro lado, diminuindo o valor de m, o número de colisões aumenta, assim como o número de elementos a serem tratados em tempo de consulta. Consequentemente, o desempenho das consultas diminui.

O processo de indexação de um conjunto de dados S usando LSH é resumido no Algoritmo 2.1. Cada elemento x do conjunto de elementos S é projetado usando m funções *hash* que transformam x em m números reais (x') (o vetor *hash*). Esses m números são então quantificados para um único número (g), a fim de projetar os objetos semelhantes no mesmo *bucket*.

Algoritmo 2.1: Algoritmo de construção para LSH

1 **Entrada**: O conjunto de dados S
2 **Saida**: Todos os objetos mapeados nas L tabelas $hash$
3   **for** $i = 1$ **to** L **do**
4     $H_i \Leftarrow \{h_1, ..., h_m\}$ *//Inicializar tabela T_i com um conjunto de funções hash H_i*
5   **end for**
6   **for** $i = 1$ **to** L **do**
7   **foreach** x em S **do**
8       $x' \Leftarrow < h_1(x), ..., h_m(x) >$ *//Projetar x: m vezes usando o conjunto de funções hash H_i*
9       $g \Leftarrow quantize(x')$ *//Processo de quantização − calcular o valor hash final*
10       $I \Leftarrow T_i[\, g \bmod |T_i|\,]$ *//localizar o bucket I*
11   adicionar a entrada $< x, g >$ em I armazenar a referência de x e o valor g
12   **end for**
13   **end for**

Uma vez que cada *bucket* é definido em termos de similaridade (isto é, elementos semelhantes tendem a ser encontrados no mesmo *bucket*), é provável encontrar os vizinhos mais próximos a um objeto q nos *buckets* recuperados por cada conjunto de funções *hash*. Assim, ao invés de fazer comparações para reduzir o espaço de busca, o LSH mapeia diretamente os objetos aos *buckets*. Logo, é possível obter custo sublinear nas consultas. Numa consulta kNN, a fim de retornar os k vizinhos mais próximos, apenas as distâncias entre q e os elementos dentro dos *buckets* são calculadas. Numa consulta por abrangência acontece o mesmo, mas a condição da consulta é diferente e só os elementos dentro do raio de consulta são retornados. O algoritmo de consulta por abrangência aproximada é descrito no Algoritmo 2.2.

Algoritmo 2.2: Consulta por abrangência aproximada usando LSH

1 **Entrada**: O objeto de consulta q e o raio da consulta r
2 **Saida**: Os objetos que satisfazem às condições de consulta.
3   **for** $i = 1$ **to** L **do**
4     $q' \Leftarrow < h_1(q), ..., h_m(q) >$ *//Projetar x: m vezes usando o conjunto de funções hash H_i*
5     $g \Leftarrow quantize(q')$ *//Processo de quantização − calcular o valor hash final*
6     $I \Leftarrow T_i[\, g \bmod |T_i|\,]$ *//localizar o bucket I*
7   **foreach** e em I **do**
8   **if** $e.g = g \wedge d(q, e.x) \leq r$ **then**
9   retornar $e.x$ se não foi retornado antes
10   **end if**
11   **end for**
12   **end for**

O algoritmo kNN pode ser obtido por meio de simples modificações no Algoritmo 2.2. Basicamente, o algoritmo kNN terminará quando encontrar k objetos distintos ou se não houver mais *buckets* para explorar. No entanto, é importante lembrar que LSH garante resultados de qualidade previsível apenas para consultas por abrangência aproximadas $((1+\epsilon) - Rq(q,r))$. Além disso, alguns inconvenientes do LSH não foram resolvidos por completo, por exemplo: (1) LSH requer vários subíndices de tal forma que cada subíndice organiza o conjunto de dados inteiro usando uma tabela *hash* com funções *hash* independentes. Essa exigência é extremamente crítica para melhorar a precisão da busca, mas leva a um alto consumo de memória; (2) LSH tem uma dependência crítica dos parâmetros de domínio, que determinam o número de funções *hash* e o número de tabelas *hash*.

Em [Lv et al., 2007] os autores propuseram o LSH Multi-probe, que visa manter o custo de memória aceitável. O LSH Multi-probe é baseado no LSH clássico, mas enquanto o algoritmo de consulta LSH examina apenas um *bucket* para cada tabela *hash*, o LSH Multi-probe examina os *buckets* que são susceptíveis de conter os resultados da consulta por tabela *hash*. Consequentemente, o número de tabelas *hash* é reduzido sem perda significativa de precisão. Intuitivamente o LSH Multi-probe verifica *buckets* de forma mais inteligente, gerando τ $probes$ por tabela *hash*, o que permite explorar τ *buckets* por subíndice, e como consequência, aumentar o número de candidatos sem incrementar o número de subíndices.

Uma pesquisa recente sobre o LSH [Dong et al., 2008] mostra que o desempenho nas consultas não só depende da distribuição geral dos conjunto de dados, mas também da distribuição local em torno ao objeto de consulta. No LSH Multi-probe, um número fixo de $probes$ pode ser insuficiente para algumas consultas e maior do que o necessário para outras. No entanto, ainda é complicado ajustar o número de funções *hash* associado ao raio de consulta.

Dado que seus parâmetros também dependem do número de objetos a serem indexados, o LSH não é uma solução incremental. Para lidar com esse problema, uma nova proposta foi apresentada, o LSH-Forest [Bawa et al., 2005]. Essa técnica foi desenvolvida para suportar auto ajuste em relação ao número de funções *hash*. Essencialmente, o LSH-Forest é uma coleção de árvores prefixo onde cada uma pode ter um número diferente de funções *hash*. No entanto, ainda é necessário ajustar o número de subíndices, ou seja, o número de árvores prefixos. Além disso, não está claro o seu desempenho em sistemas que trabalham com dados distorcidos, em que tende-se a verificar mais candidatos do que o necessário em pequenas áreas densas e ter candidatos insuficientes para pontos de consulta em áreas esparsas.

Nesse mesmo cenário, a técnica chamada LSH Multi-level [Ocsa e Sousa, 2010], desenvolvida no contexto deste trabalho de mestrado, propõe uma nova abordagem de *hashing* para resolver alguns desses problemas. Especificamente, utilizando uma abordagem multirresolução para resolver a dependência de parâmetros de domínio. O método não espera os parâmetros do domínio de dados, pois, se adapta dinamicamente durante o processo de indexação, graças às habilidades auto-adaptativas da estrutura do índice multinível. O LSH Multi-level apresenta um desempenho superior em termos de tempo e espaço comparado com as outras abordagens *hash*, pois a técnica multinível redistribui os objetos em *buckets* de maneira uniforme em todos os níveis. Isto porque, em contraste com o LSH, o LSH Multi-level explora a estrutura multirresolução para calcular e localizar os vetores *hash* apropriados para uma consulta específica. Assim, vários vetores *hash* de diferentes resoluções são computados por cada índice no processo de consulta e, como consequência, não se precisa de mais índices para garantir resultados da mesma qualidade.

Recentemente, em [Tao et al., 2010] os autores propuseram o LSB-Forest. Esse trabalho melhora a técnica LSH-Forest, e a fim de garantir a qualidade e eficiência de recuperação de dados multidimensionais, a técnica utiliza representações baseadas em *space-filling curves* e árvores B. Assim, os valores *hash* são representados como valores unidimensionais usando tanto a projeção LSH como o das curvas Z. Além disso, várias árvores B são usadas para indexar os dados com o objetivo de melhorar a qualidade e eficiência dos resultados. A desvantagem da àrvore LSB é que usa leituras/escritas randômicas, o que requer um número considerável de acessos a disco quando o banco de dados é grande. Para resolver esse problema, o HashFile [Zhang et al., 2011] foi proposto. Como detalhado na seção seguinte, em comparação aos atuais métodos LSH, o HashFile apenas divide recursivamente os *buckets* densos para atingir partições mais equilibradas. Cada *bucket* armazena um número fixo de objetos, mas o HashFile aproveita a varredura sequencial em vez de usar acesso randômico, como é o caso nas árvores B por exemplo. O HashFile é baseado no conceito de projeção aleatória (*Random Projection*).

2.5.3 HashFile

Recentemente, o HashFile [Zhang et al., 2011] foi proposto para fornecer consultas kNN exatas no espaço L_1 e consultas aproximadas no espaço L_2. O método combina as vantagens da projeção aleatória e a varredura sequencial. Ao contrário dos outros métodos baseados no LSH, em que cada *bucket* é associado a uma concatenação de m valores *hash*, o HashFile apenas particiona recursivamente os *buckets* densos que são organizados em uma estrutura hierárquica. Note que ao contrario da maioria dos métodos LSH, o HashFile só precisa de um subíndice para garantir resultados de qualidade. Assim, dado um objeto de consulta q, o algoritmo de busca só explora os *buckets* que estão perto ao objeto de consulta usando a abordagem *top-down*. Os *buckets* candidatos em cada nó são armazenados sequencialmente em ordem crescente ao valor *hash* e podem ser eficientemente carregados na memória com apenas uma varredura linear. Os resultados mostram que o HashFile supera métodos recentes, que representam o estado de arte em métodos de indexação, para consultas kNN exatas e aproximadas.

Projeção Aleatória de Dados

A projeção aleatória é um método de redução da dimensionalidade computacionalmente eficiente e suficientemente preciso. Dado um conjunto de dados S com N pontos de dimensionalidade n e uma matriz aleatória $R_{n\times k}$, a projeção é calculada da seguinte forma:

$$S'_{N\times k} = S_{N\times n} \times R_{n\times k} \tag{2.9}$$

A projeção resulta em um conjunto de dados S' k-dimensional com N pontos. A projeção aleatória pode preservar a distância Euclidiana no espaço reduzido, o que é especificado pelo Lema Jonhson-Lindenstrauss.

Lema de Jonhson-Lindenstrauss: Dado $\epsilon > 0$ e um inteiro N, seja k um inteiro positivo tal que $k \geq k_0 = O(\epsilon^{-2} logN)$. Para cada conjunto S de N pontos em ^{n}, existe $f : {}^n \to {}^k$ tal que para todo $u, v \in S$, tem-se:

$$(1-\epsilon) \parallel u - v \parallel^2 \leq \parallel f(u) - f(v) \parallel^2 \leq (1+\epsilon) \parallel u - v \parallel^2 \tag{2.10}$$

Nos últimos anos, o Lema de Jonhson-Lindenstrauss (JL) tem sido útil na resolução de uma variedade de problemas. A ideia é a seguinte: ao fornecer uma baixa representação dimensional dos dados, JL acelera certos algoritmos de forma significativa, em especial algoritmos cujo tempo de execução depende exponencialmente na dimensão do espaço de trabalho (há uma série de problemas práticos para os quais os algoritmos mais conhecidos têm esse comportamento). Ao mesmo tempo, JL provê uma garantia de manutenção da proximidade depois de uma operação de projeção sobre cada par de distâncias e geralmente é suficiente para estabelecer que a solução encontrada no espaço reduzido é uma boa aproximação da solução ótima no espaço original.

Alguns exemplos de aplicação foram apresentado em [Indyk e Motwani, 1998]. Por exemplo, Indyk and Motwani mostraram que o Lema JL é útil para resolver o problema de busca aproximada ao vizinho mais próximo, onde depois de alguns procedimentos de pré-processamento do conjunto de dados, consultas desse tipo são respondidas: Dado um ponto arbitrário x e um ponto $y \in P$, para cada ponto $z \in P$ se satisfaz $\|x - z\| \geq (1 - \epsilon)\|x - y\|$

Restrição da distância para consulta kNN usando L_1

Suponha que $\mathcal{H}$ é uma função *hash* derivada de qualquer linha $R_{n\times k}$ tal que esta mapeia um objeto o de dimensão n em um valor unidimensional.

$$\mathcal{H}(o) = \lfloor \Sigma_{i=1}^{n} h_i \cdot o_i \rfloor \tag{2.11}$$

Dado que cada elemento h_i está dentro do $\mathcal{H}$ para valores randômicos $\{-1, 0, 1\}$, pode-se facilmente atingir um *lower bound* para consultas com a distância L_1.

Limite Inferior para L_1 Dados dois pontos x e y de dimensão n, e uma função *hash* $\mathcal{H} : {}^n \to {}^1$ com $h_i \in \{-1, 0, 1\}$, tem-se.

$$\| x - y \|_{L_1} \geq |\mathcal{H}(x) - \mathcal{H}(y)| - 1 \tag{2.12}$$

Assim, a projeção aleatória pode ser usada para particionar um grande volume de dados em *buckets*, em um espaço unidimensional. As páginas no disco podem ser armazenadas sequencialmente em ordem crescente ao valor *hash*. Assim, dado um ponto de consulta q e a distância λ ao vizinho mais próximo já encontrado, se o valor *hash* de q é h_q, então de acordo com o relação apresentada na Equação 2.12, apenas as páginas com valor *hash* entre $[h_q - \lambda, h_q + \lambda]$ precisam ser acessadas. Os demais pontos fora do intervalo podem ser seguramente podados.

Restrição da distância para consulta kNN approximada usando L_2

Limite Inferior para L_2 Dado dois pontos x e y de dimensão n, e uma função *hash* $\mathcal{H}_w(o) = \lfloor \frac{\vec{h_i} \cdot \vec{o_i}}{w} \rfloor$, tem-se.

$$\| x - y \|_{L_2} \geq (1 - \epsilon) w (|\mathcal{H}(x) - \mathcal{H}(y)| - 1) \tag{2.13}$$

Basicamente para poder reduzir os dados em valores unidimensionais, a Equação 2.8 é adaptada para trabalhar como uma extensão da projeção aleatória. As páginas no disco também podem ser armazenadas

sequencialmente em ordem crescente ao valor *hash*. Assim, dado um ponto de consulta q e a distância λ ao vizinho mais próximo já encontrado, se o valor *hash* de q é h_q, então de acordo com o relação apresentada na Equação 2.13, as páginas com valor *hash* entre $[h_q - \lambda, h_q + \lambda]$ são uma boa aproximação dos dados que precisam ser acessadas.

Construção do índice

O HashFile é construído seguindo a abordagem *top-down*. No início, uma função *hash* é gerada no nó raiz e cria um arquivo no disco para armazenar os dados. Dado que não é possível predizer a gama de valores *hash*, não se pode atribuir uma coleção de páginas de tamanho fixo com antecedência. Inicialmente, há apenas uma página alocada com um intervalo (∞, ∞). O tamanho da página é fixo B, indicando que ela pode acomodar até B pontos. Os primeiros B objetos podem ser inseridos com sucesso na página. Quando o $(B+1)$-th objeto chega, a página é dividida usando um valor *hash* h_s que divide a página em duas novas páginas. Agora, o novo objeto pode ser inserido na nova página com base em seu valor *hash*. Como novos objetos são inseridos de forma contínua, uma operação de divisão ocorre e os intervalos *hash* tornam-se menores. Finalmente, um objeto pode ser mapeado para um *bucket* cheio onde todos os pontos têm o mesmo valor *hash* e portanto não pode ser dividido. Para inserir esse novo objeto, cria-se um nó filho com uma nova função *hash*. Os pontos nessa página são extraídos e são remapeados junto com o novo objeto no nó filho.

A Figura 2.11 ilustra um exemplo da estrutura da árvore do HashFile, assim como a lógica de um nó interno da árvore. A árvore é construída pela inserção contínua de pontos de dados. Quando um *bucket* no nó pai não pode ser dividido mais, nós filhos são criados para particionar o *bucket* mais denso. Portanto, cada nó interno contém uma lista de nós filhos, como é ilustrado na figura do exemplo. Cada bloco representa uma página com um intervalo *hash* $[l_i, h_i]$.

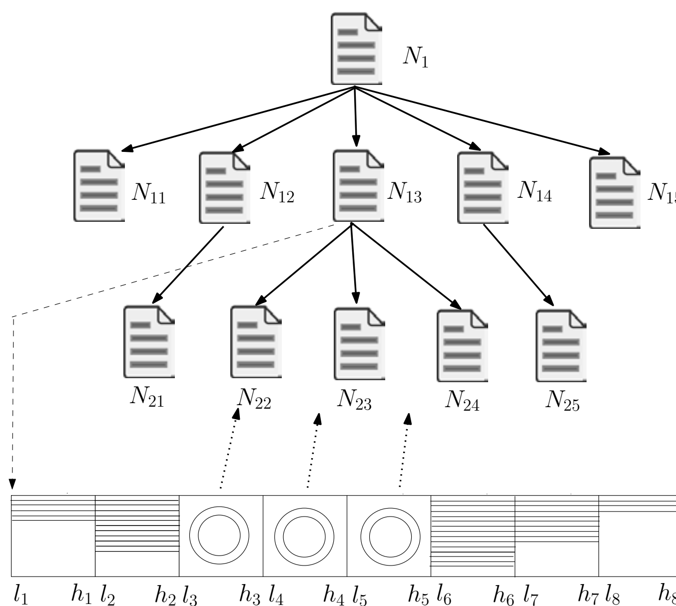


Figura 2.11: Estrutura do nó no HashFile. Figura adaptada de [Zhang et al., 2011].

Consulta kNN Aproximada

No LSH, cada tabela *hash* é associada a um conjunto de m funções *hash*. Intuitivamente, pode-se considerar que todos os *buckets* estão organizados em uma árvore com altura equilibrada em que o comprimento do caminho desde a raiz até os nós folha é sempre m. Quando m aumenta, o tamanho do *bucket* torna-se menor e é mais provável que objetos próximos uns dos outros sejam mapeados em *buckets* vizinhos e portanto não sejam cobertos pelo algoritmo de busca. Assim, ampliando o espaço de busca, inspecionando *buckets* vizinhos pode-se melhorar significativamente a qualidade dos resultados [Lv et al., 2007].

Inspirados por esta idea, os autores do HashFile propuseram um método flexível e eficaz para o processo de consulta kNN. Os usuários podem especificar um parâmetro λ para determinar o espaço de busca. Se o objeto de consulta q é mapeado para o valor *hash* h_q, só é necessário explorar os *buckets* vizinhos que tenham interseção com o intervalo *hash* $[h_q-\lambda, h_q+\lambda]$. Note que λ especifica o intervalo de consulta em vez do número de *buckets* vizinhos. No exemplo ilustrado na Figura 2.12 o objeto de consulta q é mapeado no *bucket* B_5 como valor 9. Quando $\lambda = 1$, os *buckets* B_4, B_5, B_6 são explorados. Todos os pontos no *bucket* B_6 são processados sequencialmente. Dado que os *buckets* B_4 e B_5 corresponder a nós filhos, eles são processados na mesma forma que o nó pai. Para $\lambda = 3$, os *buckets* $B_2, B_3, B_4, B_5, B_6, B_7$ são explorados. Dado que os *buckets* B_2, B_6, B_4 são armazenados sequencialmente, eles podem ser carregados na memoria principal e logo usar a varredura linear. O pesudocódigo para responder consultas kNN aproximada é apresentado no Algoritmo 2.3.

Algoritmo 2.3: Consulta kNN aproximada usando HashFile

```
1  algorithm ApproximateNN(q, λ)
2      init δ
3      δ := ApproximateNNInNode(q, root, δ, λ)
4      return δ
5  end.
6  procedure [δ] = ApproximateNNInNode(q, node, δ, λ)
7      h_q := node.hash(q)
8      for cada nó child do
9          wdist := |child.hash_value − h_q|
10         if wdist < λ then
11             inserir child no heap ordenado pelo wdist
12         end if
13     end for
14     for cada página em disco p_i do
15         buscar a lista de páginas de p_start a p_end no disco entre os intervalor [h_q − λ, h_q + λ]
16     end for
17     carregar as páginas entre [h_q − λ, h_q + λ] no bloco B
18     for cada objecto o em B do
19         dist := ||o − q||_L2
20         if dist < δ then
21             δ := dist
22         end if
23     end for
24 end.
```

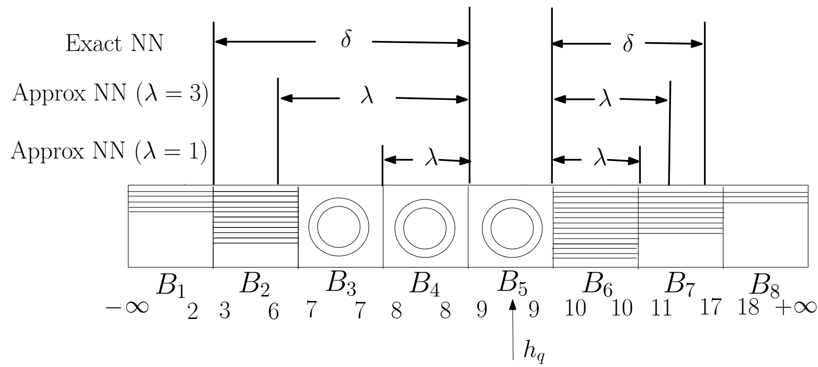


Figura 2.12: Busca kNN aproximada no nó do HashFile. Figura adaptada de [Zhang et al., 2011].

2.6 Considerações Finais

Como discutido ao longo deste capítulo, soluções exatas para resolver o problema de busca por similaridade em altas dimensões vêm sendo estudadas há vários anos, enquanto algoritmos probabilísticos têm sido pouco explorados. De acordo com a literatura, é eficiente executar busca kNN exata para dados de baixa dimensionalidade, mas em dimensões altas as soluções aproximadas são também aceitáveis. O equilíbrio entre a eficiência e eficácia torna-se progressivamente mais crítico conforme a dimensionalidade dos dados cresce, devido a um fenômeno conhecido como "maldição da alta dimensionalidade".

Muitas técnicas exatas e aproximadas, como os Métodos de Acesso Métricos (MAMs) e os métodos baseados em *Space-filling curves*, *Locality Sensitive Hashing* (LSH) e suas extensões, foram propostas para resolver busca por similaridade em altas dimensões. Entre elas, os métodos baseados no LSH são das poucas técnicas com garantia teórica de custo sublinear. No entanto, muitas de suas implementações atuais apresentam dificuldades para ser implantados em problemas reais, como por exemplo, tarefas de mineração de dados que dependem de algoritmos de busca ao vizinho mais próximo. Logo, o potencial de aprimoramento de abordagens aproximadas para busca kNN é importante para transformar algoritmos clássicos de mineração de dados em algoritmos eficientes para domínios complexos.

Mesmo considerando soluções aproximadas como abordagem para acelerar consultas por similaridade, problemas como a busca AllkNN em altas dimensões ainda são consideradas impraticáveis devido a seu alto custo computacional. Para desenvolver soluções de alto desempenho para tarefas que demandem grandes recursos de computação, precisa-se a utilização de novos esquemas de programação. Uma abordagem para atacar essas restrições e melhorar o tempo de execução de técnicas de recuperação de dados é empregar técnicas de computação GPGPU, como descrito no Capítulo 5.

CAPÍTULO 3

Mineração de Séries Temporais

3.1 Considerações Iniciais

Os rápidos avanços nas tecnologias de coleta e de armazenamento permitiram o acúmulo de grandes quantidades de dados. No entanto, extrair conhecimento provou ser extremamente desafiador. Neste contexto, a mineração de dados combina métodos tradicionais de análise de dados com algoritmos sofisticados para o processamento de grandes volumes de dados [Tan et al., 2005].

A Mineração de Dados (*Data Mining*) é uma área de pesquisa inserida no contexto mais amplo de Descoberta de Conhecimento em Bases de Dados (KDD - *Knowledge Discovery in Databases*), cujo objetivo principal é extrair de um conjunto de dados o conhecimento a ser utilizado em processos decisórios. Mais especificamente, os principais objetivos dos métodos de mineração de dados são a descrição de um conjunto de dados através de descoberta de padrões que aparecem regularmente nos dados e a predição de valores futuros de interesse baseada em conhecimento prévio de um banco de dados [Fayyad et al., 1996].

A Mineração de Séries Temporais é um campo relativamente novo, que inclui técnicas de Mineração de Dados adaptadas para levar em consideração a natureza das séries temporais. Em vários domínios de aplicação, como nos negócios, indústria, medicina e ciência, os procedimentos geram grandes quantidades de dados caracterizadas como séries temporais. Além disso, essas técnicas podem ser aplicadas em dados extraídos de imagem ou vídeo, pois esses tipos de dados podem ser considerados também como séries temporais.

De acordo com literatura da área, as principais tarefas em mineração de séries temporais são: identificação de agrupamentos, classificação, identificação de intrusos, descoberta de *motifs*, descoberta de regras de associação e previsão. Embora algumas dessas tarefas sejam semelhantes às tarefas correspondentes de mineração de dados, o aspecto temporal coloca algumas questões específicas a serem consideradas e/ou restrições impostas nas aplicações correspondentes. Primeiro, como tarefas e algoritmos de mineração podem ser baseados em buscas por similaridade, torna-se necessária a utilização de medidas de similaridade entre séries temporais, esta questão é muito importante na mineração de

séries temporais, pois envolve um certo grau de subjetividade que pode afetar o resultado final de tarefas e algoritmos que utilizam a busca por similaridade como parte do processo de mineração. A segunda questão, relacionada à medida de similaridade, é a representação de séries temporais visando redução de dimensionalidade. Como a quantidade de dados pode variar de poucos *megabytes* até *terabytes*, uma representação adequada da série temporal é necessária para manipular e analisar os dados de maneira eficaz.

Na Seção 3.2, são apresentados métodos de representação de séries temporais. Na Seção 3.3, discute-se trabalhos relacionados a medidas de similaridade em séries temporais. Na Seção 3.4, discute-se trabalhos relacionados a consultas por similaridade em séries temporais. Na Seção 3.5 são discutidos os principais conceitos de Mineração de Dados e Descoberta de Conhecimento em Bases de Dados (KDD), e como estes conceitos são adaptados para levar em consideração a natureza das séries temporais. Na seção 3.6 são descritos em profundidade algumas das principais propostas para resolver a descoberta de *motifs*, que junto com as consultas por similaridade são as atividades mais relevantes para este trabalho.

3.2 Representação de Séries Temporais

Uma série temporal é uma coleção de observações feitas sequencialmente ao longo do tempo. Em cada ponto de medição no tempo podem ser monitorados um ou mais atributos, e a série temporal resultante é chamada univariada ou multivariada, respectivamente. Em muitos casos, uma sequência de símbolos pode ser usada para representar uma série de temporal.

Uma série temporal $T = (T_1, T_2, ..., T_n)$ é um conjunto ordenado de n valores reais. As séries temporais podem ser longas, por vezes contendo milhares de milhões de observações. No entanto, usualmente o interesse não está nas propriedades globais da série, mas sim, nas subpartes das séries, as quais são chamadas subsequências.

Diversas representações de séries temporais foram propostas na literatura, principalmente com a finalidade de redução de dimensionalidade. Algumas das representações mais usadas são apresentadas em [Lin et al., 2003] em um diagrama de árvore, ilustrado na Figura 3.1.

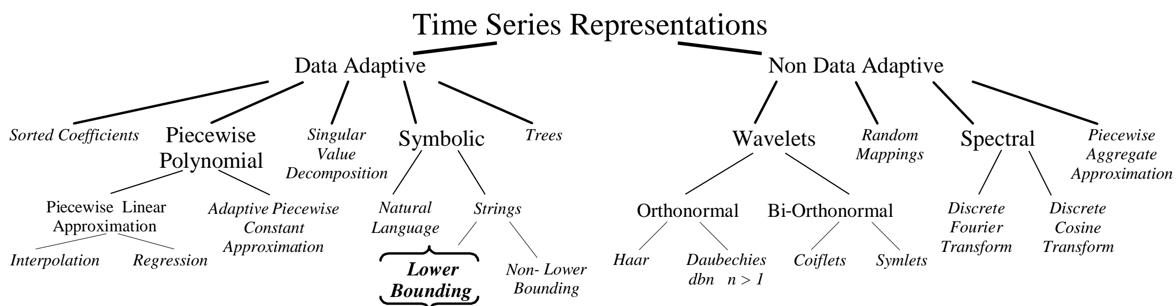


Figura 3.1: Hierarquia de representações de séries temporais. Os nós folha referem-se à representação atual, e os nós internos referem-se à classificação da abordagem [Lin et al., 2003].

A Transformada Discreta de Fourier (*DFT*) [Agrawal et al., 1993] foi uma das formas de representação proposta pela primeira vez no contexto de mineração de dados. DFT transforma uma série de temporal a partir do domínio do tempo para o domínio da frequência. A Transformada

Discreta Wavelet (*Discrete Wavelet Transform* DWT) [Chan e Fu, 1999], transforma a série temporal em espaço/frequência. O *Singular Value decomposition* (SVD) [Korn et al., 1997] realiza uma transformação global, girando o eixo do conjunto de dados de tal modo que o primeiro eixo explica a variação máxima, o segundo eixo explica o máximo da variância restante e é ortogonal ao primeiro eixo, e assim por diante. *Piecewise Aggregate Approximation* (PAA) [Keogh et al., 2001] divide uma série temporal em segmentos de igual comprimento e registra a média dos valores correspondentes de cada segmento.

Para agrupar séries temporais similares, com uma margem de ajuste variável, precisa-se de métodos de aproximação de séries temporais. Entre esses, o SAX *Symbolic Aggregate Approximation* (SAX) [Lin et al., 2007] mostrou-se como uma das melhores técnicas de representação de séries temporais para tarefas de mineração. O SAX usa o PAA como primeiro passo para a representação e, em seguida, a série resultante é transformada em uma sequência de símbolos usando as propriedades da distribuição de probabilidade normal. Um exemplo da representação discreta SAX é ilustrado na Figura 3.2. A transformação da representação PAA para uma representação discreta gera uma sequência de letras (palavras) para representar a série temporal. A principal vantagem do SAX é obter um modelo de representação, que reduz a dimensionalidade, mas preserva as características da série original. Apesar de perder algumas das informações para o cálculo da média do *Piece-wise Aggreagate Aproximação* (PAA), o SAX é robusto com relação ao ruído, exceto se variações súbitas são inerentes ao domínio da aplicação.

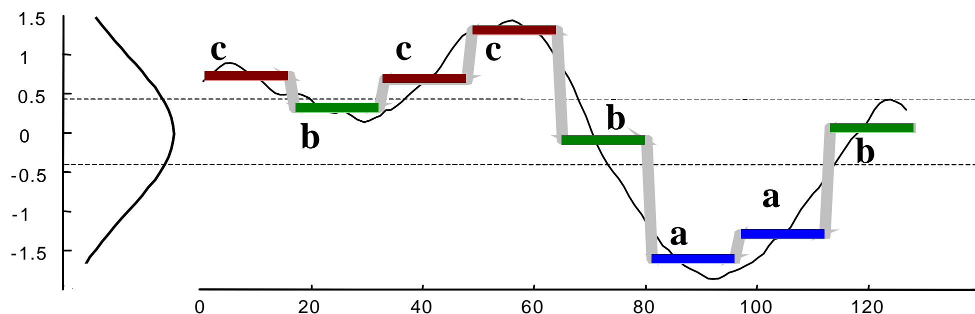


Figura 3.2: A série temporal (linha preta), já normalizada, é discretizada, obtendo-se uma representação PPA (linha cinza grossa). Depois os coeficientes PAA em letras (em negrito) são mapeados a uma representação discreta. Neste exemplo, a série é discretizada para a palavra **cbccbaab** [Lin et al., 2007].

O *indexable Symbolic Aggregate Approximation* (iSAX) [Shieh e Keogh, 2008] foi proposto como uma extenção do clássico SAX. O iSAX permite diferentes resoluções para a mesma palavra e fez o SAX original ainda mais atraente. Para evitar ambiguidade, a resolução de cada símbolo deve ser clarificado na palavra iSAX. Uma vez que a margem de semelhança é ajustável, aumentar ou diminuir a resolução iSAX torna-se um recurso disponível.

Para representar uma palavra iSAX precisa-se de representações com diferentes resoluções de uma palavra SAX. Assim, uma palavra SAX com cardinalidade a (ou resolução a) com comprimento da palavra ω de uma série temporal T é definido como:

$$SAX(T, \omega, a) = T^a = \{t_1, t_2, ..., t_\omega\} \tag{3.1}$$

Assim, com a Equação 3.1 pode-se definir palavras de diferentes resoluções, por exemplo para uma série temporal T as palavras SAX com comprimento $\omega = 4$ e resoluções $2, 4, 8, 16$ são:

$$
\begin{array}{lllll}
SAX(T,4,16) & = & T^{16} & = & \{1100, 1101, 0110, 0001\} \\
SAX(T,4,8) & = & T^{8} & = & \{110, 110, 011, 000\} \\
SAX(T,4,4) & = & T^{4} & = & \{11, 11, 01, 00\} \\
SAX(T,4,2) & = & T^{2} & = & \{1, 1, 0, 0\}
\end{array}
$$

A representação gráfica da representação iSAX com cardinalidade 4 e 2 é ilustrado na Figura 3.3.

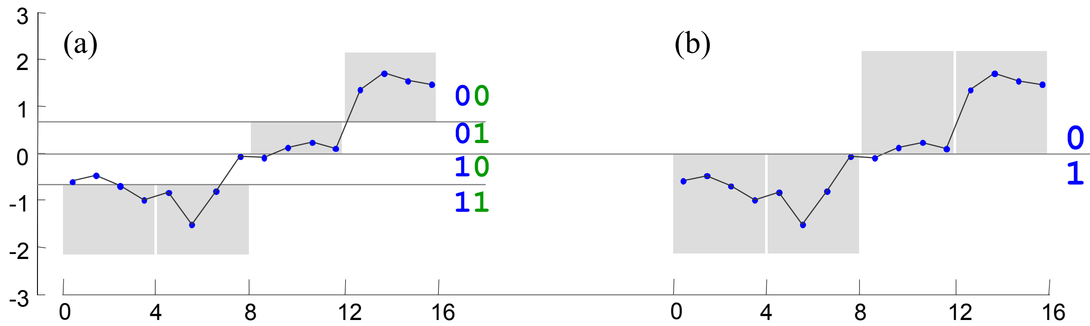


Figura 3.3: A série temporal T convertido em palavras SAX. (a) com cardinalidade 4 $\{11, 11, 01, 00\}$ (b) com cardinalidade 2 $\{1, 1, 0, 0\}$ [Shieh e Keogh, 2008].

3.3 Medidas de Similaridade em Séries Temporais

A definição de novas medidas de similaridade tem sido uma das áreas mais pesquisadas no campo da Mineração de Séries Temporais. Geralmente, as medidas de similaridade estão fortemente relacionadas com o esquema de representação aplicada aos dados originais. No entanto, existem algumas medidas de similaridade que aparecem frequentemente na literatura. A maioria das escolhas dos pesquisadores baseia-se na família Minkowski L_p, que inclui a distância Euclidiana L_2. Outra medida de similaridade que atrai muita atenção é o *Dynamic Time Warping* (DTW) [Berndt e Clifford, 1994]. A principal vantagem dessa medida é permitir a aceleração-desaceleração de uma série ao longo da dimensão temporal (alinhamentos não-lineares são possíveis). No entanto, o DTW é computacionalmente caro.

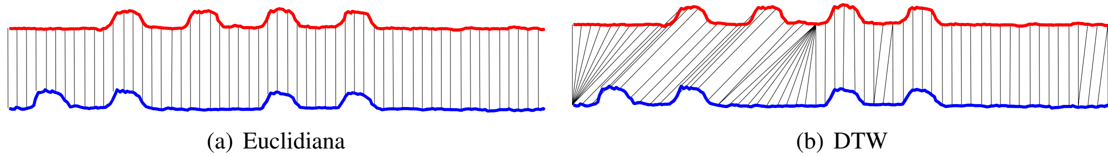
Apesar de ser simples de calcular, a distância Euclidiana produz resultados errôneos para séries temporais que são similares, mas apresentam distorções no eixo do tempo. DTW é uma medida de dissimilaridade mais eficiente nesse caso. Ao contrário da distância Euclidiana, o DTW é baseado na ideia de alinhamentos não lineares entre séries, como ilustrado na Figura 3.4.

A seguir é apresentada uma visão geral do DTW, considerando duas séries temporais, Q de tamanho n, e C de tamanho m:

$$Q = (Q_1, Q_2, ..., Q_i, ..., Q_n)$$

$$C = (C_1, C_2, ..., C_j, ..., C_m)$$

Para alinhar essas duas séries usando o DTW, constroi-se uma matriz de tamanho $n \times m$ na qual o elemento de índice (i, j) contém a distância entre as observações Q_i e C_j. Cada célula da matriz (i, j) corresponde ao alinhamento entre as observações Q_i e C_j, como ilustrado na Figura 3.5.



(a) Euclidiana (b) DTW

Figura 3.4: Apesar das duas séries terem formas similares, elas não estão alinhadas no eixo do tempo. A distância Euclidiana gera uma medida de dissimilaridade pessimista, já o DTW produz uma medida de dissimilaridade mais intuitiva devido aos alinhamentos não-lineares [Keogh, 2002].

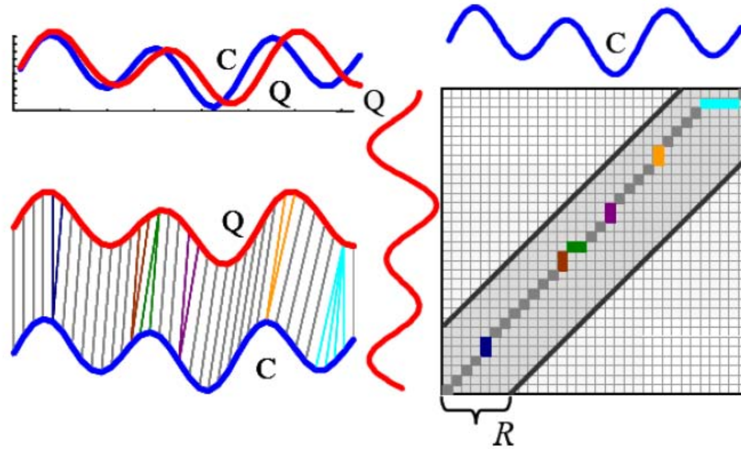


Figura 3.5: Matriz de alinhamento para o cálculo do DTW. A área em cinza representa uma janela de alinhamento [Keogh, 2002].

Um conceito importante relacionado com as medidas de similaridade e os métodos de representação de dados é o limite inferior das distâncias. Dado que o objetivo principal dos métodos de representação de séries é reduzir a dimensionalidade dos dados e, ainda assim, conservar informação suficiente para descartar rapidamente as séries temporais que não são similares, precisa-se de uma função de distância estimada para as séries temporais reduzidas.

Limite Inferior: A estimativa ($D_{estimada}$) para uma medida de similaridade entre duas séries temporais deve ser menor e o mais próxima possível da verdadeira distância ($D_{original}$) entre as séries.

$$D_{estimada}(A', B') \leq D_{original}(A, B) \tag{3.2}$$

A relação apresentada pela Equação 3.2 garante a inexistência dos falsos negativos no conjunto de resposta. Falsos negativos são séries que se qualificariam como similares segundo a distância original, mas que são distantes segundo a medida estimada. Por outro lado, falso positivo ocorre quando, na realização de uma consulta, as séries que parecem ser próximas segundo a medida estimada, na verdade não o são. Como falsos positivos podem ser removidos posteriormente, eles podem ser tolerados, contanto que não sejam muito frequentes. Com essa ideia em mente, Keogh [Keogh, 2002] definiu uma estimativa para o DTW, chamada *LB_Keogh*, e comprovou matematicamente que essa estimativa obedece à propriedade de limite inferior. O autor também comprovou, por meio de avaliações em bases de dados, que essa estimativa se aproxima melhor a distância DTW exata do que outras publicadas anteriormente.

Em [Ding et al., 2008], Ratanamahatana e Keogh, mostram que quando se utiliza a propriedade do limite inferior *LB_Keogh* com DTW, a tarefa de classificação torna-se essencialmente linear com respeito ao número total de cálculos de distância. No entanto, num trabalho mais recente, em [Ding et al., 2008] argumentam que para tarefas de classificação com bancos de séries temporais muito grandes, a distância Euclidiana pode ser tão eficaz quanto DTW, sendo a distância Euclidiana computacionalmente mais simples de calcular. Em [Ding et al., 2008], Ratanamahatana e Keogh, mostram que quando se utiliza a propriedade do limite inferior *LB_Keogh* com DTW, a tarefa de classificação torna-se essencialmente linear com respeito ao número total de cálculos de distância. No entanto, num trabalho mais recente, em [Ding et al., 2008] argumentam que para tarefas de classificação com bancos de séries temporais muito grandes, a distância Euclidiana pode ser tão eficaz quanto DTW, sendo a distância Euclidiana computacionalmente mais simples de calcular.

3.4 Consulta por Similaridade em Séries Temporais

Diversos trabalhos têm sido desenvolvidos para realizar consultas sobre séries temporais. A maioria desses trabalhos propõe algoritmos eficientes que representam os dados (séries temporais) como vetores de comprimento fixo. No entanto, diversas áreas de aplicação geram séries temporais sem restrição de comprimento e, portanto, consultas eficientes sobre essas séries constituem uma importante tarefa para recuperação de informação e também para mineração de séries.

As séries temporais de comprimento variável podem ser normalizadas para uma dimensionalidade fixa, ou seja, esticadas ou encolhidas, com alguma perda de informação, já que não se sabe a priori a dimensionalidade da série, considerando que novos dados podem ser adicionados.

A maioria dos trabalhos sobre indexação de séries temporais resolve o problema supondo que as séries têm o mesmo comprimento, ou pela execução de uma etapa de redução da dimensionalidade para normalizá-las. A Tabela 3.1 resume as principais pesquisas relacionadas considerando as seguintes características do conjunto de séries temporais: dimensão original, dimensão reduzida e método de normalização. Todos os trabalhos apresentados na tabela têm como objetivo dados com altas dimensões, mas a dimensionalidade em cada caso é reduzida para acelerar o processo em memória principal. A ideia intuitiva é que o método de redução de dimensionalidade conserva informação suficiente para descartar rapidamente as séries temporais que não são similares no processo de busca. Após identificar o conjunto de candidatos, as séries temporais (não reduzidas) são recuperadas da memória secundária para uma análise exaustiva. Este esquema, em combinação com os Métodos de Acesso Espaciais (MAEs), é a base da proposta em [Faloutsos et al., 1994], o chamado *framework GEMINI (GEneric Multimedia INdexIng)*. Diversos trabalhos estenderam o *framework* GEMINI para oferecer suporte a séries temporais [Keogh et al., 2001, Keogh, 2002, Vlachos et al., 2006, Assent et al., 2008].

Em geral, a busca por similaridade em séries temporais é baseada no casamento (*match*) da série completa. Neste caso, o objetivo da busca é encontrar, em um conjunto de séries de comprimento fixo n, as séries similares a uma determinada série de consulta q, também de comprimento n.

Numa outra abordagem, a busca por similaridade pode ser baseada em casamento de subsequências, em que a série de consulta q possui um comprimento menor que a séries do conjunto a ser consultado. Logo, o objetivo da busca é encontrar, nas séries do conjunto, subsequências similares a q. Este tipo de busca é aplicado, por exemplo, na descoberta de *motifs*.

Tabela 3.1: Trabalhos recentes em indexação de séries temporais.

Trabalho	**Dimensão original**	**Dimensão reduzida**	**Método de redução**
iSAX [Shieh e Keogh, 2008]	480, 960, 1440, 1920	16, 24, 32, 40	iSAX
TS-Tree [Assent et al., 2008]	256, 512, 1024	16, 24, 32	PAA
Scaled and Warped Matching [Fu et al., 2008]	32, 64, 128, 256, 512, 1024	21, 43, 85, 171, 341, 683	*Uniform scaling*
Exact indexing of DTW [Keogh, 2002]	32, 256, 1024	16 (todos)	PAA

3.5 KDD e Mineração de Dados

Descoberta de Conhecimento em Base de Dados (KDD) - é o processo de, a partir dos dados, identificar padrões válidos, novos, potencialmente úteis e compreensíveis. Segundo [Fayyad et al., 1996], esse processo é composto de cinco etapas: seleção dos dados; pré-processamento e limpeza dos dados; transformação dos dados; Mineração de Dados (*Data Mining*); e interpretação e avaliação dos resultados. A interação entre estas diversas etapas pode ser observada na Figura 3.6, sendo que as três primeiras podem ser interpretadas como a análise exploratória dos dados.

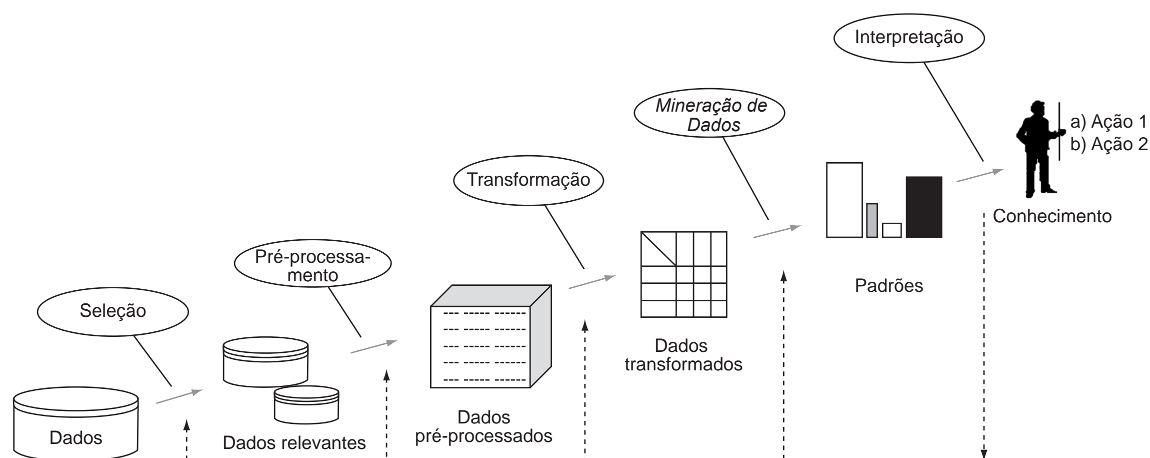


Figura 3.6: Etapas do processo KDD. Figura adaptada de [Fayyad et al., 1996].

A etapa de mineração de dados inclui a definição da tarefa de mineração a ser realizada, a escolha do algoritmo a ser aplicado e a extração de padrões de interesse. Na etapa seguinte, os padrões são interpretados e avaliados, e se os resultados não forem satisfatórios (válidos, novos, úteis e compreensíveis), o processo retorna a um dos estágios anteriores. Caso contrário, o conhecimento descoberto é consolidado [Fayyad et al., 1996]. O processo KDD refere-se a todo o processo de descoberta de conhecimento útil nos dados, enquanto a Mineração de Dados refere-se à aplicação de algoritmos para extrair modelos dos dados.

As atividades de Mineração de Séries Temporais podem ser consideradas como uma extensão das técnicas tradicionais de Mineração de Dados que leva em consideração a natureza temporal deste tipo de dado.

Na Seção 3.5 são sintetizadas as principais atividades de mineração de séries temporais, com destaque para descoberta de *motifs*, descrita em mais detalhes na Seção 3.6 por ser foco deste trabalho.

Atividades da Mineração de Séries Temporais

Conforme mencionado na Seção 3.1, as tarefas de mineração de séries temporais mais comuns são: identificação de agrupamentos, classificação, identificação de intrusos, descoberta de *motifs*, descoberta de regras de associação e previsão. Uma breve descrição de cada tarefa é apresentada a seguir.

Detecção de Agrupamento: Procurar grupos de séries temporais em um banco de dados de modo que séries temporais de um mesmo grupo sejam semelhantes umas às outras, enquanto séries temporais de grupos distintos sejam diferentes entre si.

Classificação: Atribuir uma série temporal a uma classe pré-definida de modo que a série seja mais parecida com as séries dessa classe do que com as série temporais de outras classes.

Detecção de intrusos: Procurar todas as séries temporais que contêm um comportamento diferente ao esperado com respeito a um modelo base.

Detecção de *motifs*: Procurar padrões repetidos nas séries temporais que não sejam previamente conhecidos no banco de dados.

Detecção de regras de associação: Inferir regras de uma ou mais séries temporais descrevendo o comportamento mais provável que podem apresentar em um ponto específico de tempo (ou intervalo).

Previsão: Prever eventos futuros com base em eventos passados conhecidos. Na maioria dos casos, a previsão é baseada na utilização de um modelo e de resultados das outras tarefas de mineração.

3.6 Descoberta de *motifs*

A descoberta de *motifs* é uma tarefa bem conhecida na área de bioinformática. Esse problema também despertou o interesse da comunidade de mineração de dados [Lin et al., 2002]. Os *motifs* são os padrões previamente desconhecidos que ocorrem com maior frequência nos dados. Esses padrões podem ser de particular importância para outras tarefas de mineração de séries temporais, tais como, identificação de agrupamentos, descoberta de regras de associação, identificação de anomalias e análise de comportamento. Um algoritmo eficiente para a descoberta de *motifs* também pode ser útil como uma ferramenta para sumarização e visualização de grandes volumes de dados.

As definições seguintes apresentam um resumo da terminologia, definida inicialmente em [Lin et al., 2002], usada para a definição do problema.

Banco de séries temporais: Um banco de séries temporais S é um conjunto não ordenado de N séries temporais possivelmente de diferentes comprimentos.

***R-Motif*:** Para definir o *R-motif* de um banco de séries temporais S um parâmetro de tolerância R é utilizado para decidir se duas séries temporais são suficientemente similares para serem consideradas parte do *motif*.

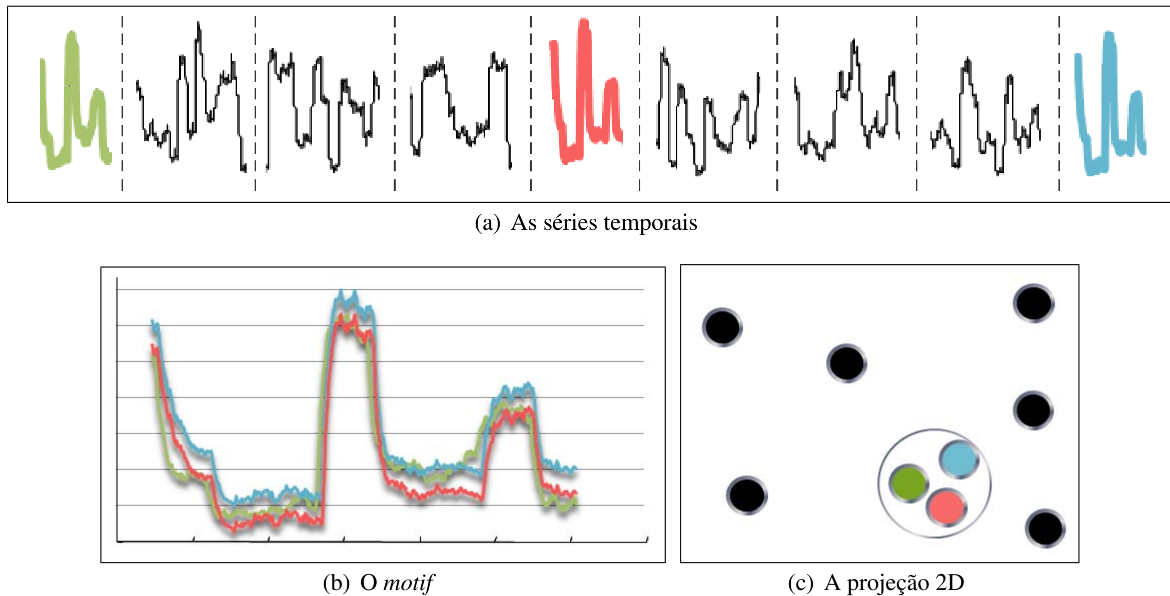
***k-Motif*:** Para definir o *k-motif* de um banco de séries temporais S são utilizados os k elementos mais próximos entre si.

***Pair-Motif*:** Esta é uma versão restrita do problema, em que se considera só as duas subsequências que estão mais próximas umas das outras, ou seja, as que apresentam a menor distância entre si. Assim, o *Pair-Motif* de um banco de series temporais S é o par não ordenado L_1, L_2 em S que são os mais similares entre todos os pares possíveis.

***Top-K^{th} Motif*:** O *Top-K^{th} Motif* de um banco de séries temporais S é o *cluster* classificado na K^th posição. Assim define-se os *TopK-Motifs* como os K padrões mais frequentes da base, ou seja os primeiros K *motifs*.

Apesar de que muitos trabalhos consideram o problema da descoberta de *motifs* como a procura dos *motifs* em subsequências de uma série temporal, esta definição tem algumas questões a resolver. Por exemplo, precisa-se definir se duas subsequências seguidas são o suficientemente diferentes para ser considerados duas subsequências independentes, tal como pode ser verificado em [Lin et al., 2002]. Assim por questões de simplicidade considera-se trabalhar apenas com bancos de séries temporais.

Intuitivamente, a noção do *R-Motif*, pode ser visto como o *cluster* mais denso em uma projeção 2D das séries. A Figura 3.7 representa esta ideia. A Figura 3.7(a) ilustra as séries temporais a ser analisadas, a Figura 3.7(b) representa o *motif* encontrado e a Figura 3.7(b) representa a projeção das séries. Note que se o *motif* é definido usando um parâmetro de tolerância R pode ser difícil saber qual é o melhor raio define o *cluster* mais denso. Pois este parâmetro é global e dependente do domínio dos dados.



(a) As séries temporais

(b) O *motif*

(c) A projeção 2D

Figura 3.7: Representação esquemática do *Motif*.

Por outro lado, se usamos a definição do *k-Motif*, em que o número de elementos do *motif* é definido pelo parâmetro k, este é menos dependente e mais simples de ajustar. Assim, o *motif* é definido pelo raio do *cluster* que contem os k elementos mais próximos entre si.

Baseado na análise anterior das definições do *R-Motif* e do *k-Motif*, daqui em diante utiliza-se a definição do *k-Motif* para definir os *TopK-Motifs*, ou seja, procura-se os padrões mais frequentes da base

conformados pelos $TopK$ primeiros *clusters*, em que cada *cluster* contém k elementos.

3.6.1 Algoritmo de força bruta

Para facilitar a descrição de um algoritmo geral de descoberta de *motifs*, primeiro considere o algoritmo de força bruta para a identificação de *motifs*. O algoritmo de força bruta, conforme descrito no Algoritmo 3.1, tem um custo que depende do custo do algoritmo de busca kNN. A versão para a procura do *Top-1 Motif* foi apresentado em [Lin et al., 2002], o Algoritmo 3.1 é uma generalização desse algoritmo para a procura exata dos *TopK-Motifs*. Assim, se o algoritmo de busca é seqüencial, a complexidade do algoritmo é quadrática. De outro modo, se o algoritmo de busca é apoiado por métodos de indexação esta complexidade pode diminuir substancialmente. Note que para que o conjunto de resposta apresente resultados relevantes, em especial para parâmetros pequenos, $k = 1$, o algoritmo kNN precisa procurar os k vizinhos mais próximos sem considerar s_i como parte do conjunto de resposta.

Algoritmo 3.1: Algoritmo de força bruta para a procura exata dos *TopK-Motifs*

```
1  algorithm [motifs] = TopKMotif(S, topK, k)
2    rangeK := ∞;
3    for i := 1 to N do
4      query_result := kNN(s_i, k)
5      dist := distance(query_result) // calcular raio do cluster
6      if dist < rangeK then
7        motifs.push(dist, query_result)
8        if motifs.size() ≥ topK then
9          motifs.cut(topK) //corte se tem mais de topK motifs
10         rangeK = distance(motifs[topK])
11       end
12     end if
13   end for
14 end.
```

O algoritmo mantém os melhores candidatos de *motifs*, assim como seu raio em cada iteração. Essas variáveis são atualizadas conforme o algoritmo encontra *clusters* com raios menores. Portanto, o *motif* é o *cluster* com o menor raio. Note que o algoritmo de busca aos k-vizinhos mais próximos é utilizado para gerar os *clusters* candidatos.

Algoritmo 3.2: Algoritmo de força bruta para a procura exata do *Pair-Motif*

```
1  algorithm [L1, L2] = PairMotif(S)
2    bsf := ∞;
3    for i := 1 to N do
4      for j := j + 1 to N do
5        dist := distance(s_i, s_j)
6        if dist < bsf then
7            bsf := dist, L1 := i, L2 := j
8        end if
9     end for
10   end for
11 end.
```

Para questões práticas mostra-se também o pseudocódigo para a procura do *Pair-Motif* [Mueen et al., 2009] usando também a abordagem da força bruta (Algoritmo 3.2). Note que o Algoritmo 3.1 retorna a mesma resposta do que o Algoritmo 3.2 para valores de $topK = 1$ e $k = 1$.

Como detalhado nas seções seguintes, devido à complexidade quadrática do algoritmo exato, os pesquisadores em [Mueen et al., 2009] têm restringido a definição do *motif* ao *Pair-Motif*. Outras abordagens foram focadas em soluções aproximadas [Chiu et al., 2003, Castro e Azevedo, 2010], como por exemplo, o algoritmo baseado em projeção aleatória e o *MrMotif*.

3.6.2 Algoritmo MK

Para o problema do *Pair-Motif*, o algoritmo MK [Mueen et al., 2009] foi o primeiro algoritmo tratável e exato de descoberta de *motif*. Este algoritmo é até três vezes mais rápido do que o algoritmo de força bruta. O algoritmo MK é baseado no descarte rápido de cálculos de distância quando a soma cumulativa é maior do que a melhor suma cumulativa encontrada. A busca do *motif* é guiada por heurísticas baseadas na ordenação linear das distâncias de cada objeto a um conjunto de pontos de referência escolhidos aleatoriamente. No entanto, o uso da distância Euclideana em dados crus pode aumentar o problema da robustez do algoritmo quando se trabalha com dados com ruído, pois a distância Euclidiana pode ser altamente afetada pelo ruído [Chiu et al., 2003].

Algoritmo 3.3: Algoritmo MK para a procura exata do *Pair-Motif*

1 **procedure** $[L_1, L_2] = MK(S)$
2 $best_so_far := \infty$
3 **for** $i := 1$ **to** $nRefs$ **do**
4 $ref_i :=$ escolher uma série temporal C de S randomicamente
5 **for** $j := 1$ **to** N **do**
6 $Dist_{i,j} := d(ref_i, D_j)$
7 **if** $Dist_{i,j} < best_so_far$ **then**
8 $best_so_far := Dist_{i,j}$, $L_1 := i, L_2 := j$
9 S_i= standard_deviation($Dist_i$)
10 Encontrar a ordenação Z dos índices para a série temporal em ref tal que $S_{Z(i)} \geq S_{Z(i+1)}$
11 Encontrar a ordenação I dos índices das séries temporais em S tal que $Dist_{Z(1),I(j)} = Dist_{Z(1),I(j+1)}$
12 $offset = 0, abandon = false$
13 **while** $abandon = false$
14 $offset = offset + 1, abandon = true$
15 **for** $j := 1$ **to** N **do**
16 $reject := false$
17 **for** $i := 1$ **to** $nRefs$ **do**
18 $lower_bound := |Dist_{Z(i),I(j)} - Dist_{Z(i),I(j+offset)}|$
19 **if** $lower_bound > best_so_far$ **then**
20 $reject := true, break$
21 **else if** $i = 1$
22 $abandon := false$
23 **if** $reject = false$ **then**
24 **if** $d(S_{I(j)}, S_{I(j+offset)}) < best_so_far$ **then**
25 $best_so_far := d(S_{I(j)}, S_{I(j+offset)})$
26 $L_1 := I(j), L_2 := I(j + offset)$
27 **end**.

Para cada série temporal de referência no algoritmo MK (Algoritmo 3.3), a distância de todas as séries são calculadas. As séries temporais de referência com maior desvio padrão é usada para ordenar as séries temporais conforme as distâncias crescem. As distâncias entre as séries temporais, tal como é definido, é o *lower bound* das distâncias calculadas e é utilizado para descartar candidatos com distâncias acima desse valor. No nível superior, o algoritmo tem iterações com valores crescentes de deslocamento ($offset$), empezando com $offset = 1$. Em cada iteração, ele atualiza as variáveis com o melhor candidato encontrado até não encontrar mais.

3.6.3 Descoberta de Motifs usando *Random Projection*

Devido à complexidade quadrática de algoritmos exatos, os pesquisadores têm focado em soluções aproximadas. Estas apresentam, em geral, a solução $O(n)$ ou $O(n \log n)$ com elevados fatores constantes [Mueen et al., 2009]. Os primeiros trabalhos que consideraram essa abordagem [Chiu et al., 2003, Buhler e Tompa, 2001], são baseados em pesquisa para descoberta de padrões em sequências de DNA da comunidade de bioinformática. Os autores desenvolveram um algoritmo para encontrar *motifs* usando projeção aleatória. O algoritmo de projeção aleatória (*Random Projection*) basicamente usa vetores aleatórios provenientes de uma distribuição estável (*landmarks*). Após um passo de discretização, utilizando a representação SAX [Lin et al., 2007], as sequencias são projetadas no espaço gerado por esse vetores aleatórios.

Algoritmo 3.4: Algoritmo baseado em projeção aleatória para a procura aproximada do *TopK-Motifs*

```
1  algorithm RandomProjection(S)
2    W := BuildSAXWords(S)
3    MC := Initialize(N, 0)
4    for i = 1 to iterations do
5      BuildMatrix(W, MC)
6    end for
7    print_motifs(MC);
8  end.
9  procedure BuildMatrix(W, MC)
10   mask := random(mask_size)
11   foreach w_i in W do
12     foreach w_j in W do
13       if i ≠ j ∧ w_i[mask] = w_j[mask] then
14         MC[i, j] + +
15       end if
16     end for
17   end for
18 end.
```

O algoritmo de projeção aleatória é descrito no Algoritmo 3.4. Esta técnica consiste em agrupar as instâncias projetadas que apresentam os padrões mais frequentes (os *motifs*), seguido de um processo de refinamento. O algoritmo é robusto contra ruído e utiliza uma abordagem probabilística e iterativa, conseguindo reduzir significativamente o custo computacional da busca com uma pequena perda de precisão. A natureza aleatória da técnica permite uma comparação eficiente de sequências muito longas para a descoberta de características relevantes. O algoritmo usa uma matriz de colisão como estrutura

base cujas linhas e colunas são a representação SAX de cada subsequência da série temporal. A eficiência em espaço do algoritmo depende em como é implementada a matriz de colisão. Aliás, o algoritmo assume que a matriz de colisão pode caber na memória principal, o que em alguns cenários não é o caso. O algoritmo depende de muitos parâmetros que precisam ser afinados, o comprimento do *motif*, o número de colunas na matriz de colisão, o tamanho do alfabeto e o tamanho da palavra da técnica de representação de dados SAX. Embora alguns dos parâmetros sejam fáceis de otimizar, na maioria de vezes o algoritmo não é aplicável a grandes conjuntos de dados. Por isso, em muitos casos, não se consegue atingir os valores dos parâmetros ideais, o que pode levar a resultados enganosos: *motifs* não encontrados, um grande número *motifs* ou *motifs* sem sentido.

O algoritmo de projeção aleatória explora o método de representação de séries temporais SAX. A ideia é que elementos com a mesma codificação SAX têm uma alta probabilidade de serem altamente semelhantes no espaço original. Como podemos ver na Figura 3.8, a representação SAX aproxima e capta semelhanças entre séries temporais do exemplo na Figura 3.7. Este fato é o fundamento de muitas pesquisas, incluindo o algoritmo *MrMotif*, como detalhado na próxima seção.

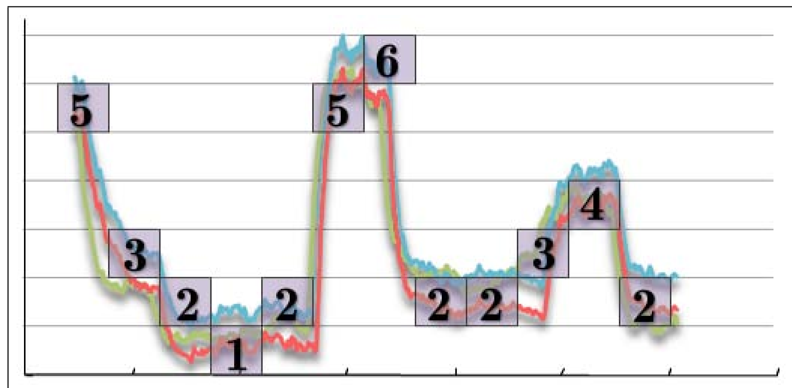


Figura 3.8: As séries temporais que definem o *motif* na Figura 3.7 compartilhan a mesma palavra SAX.

3.6.4 *MrMotif*

Recentemente, Castro introduziu outro algoritmo aproximado para a descoberta de *motifs* chamado *MrMotif* [Castro e Azevedo, 2010]. O algoritmo modela o problema como a descoberta dos K padrões mais frequentes na base. Os autores usam o método de representação de séries temporais iSAX [Shieh e Keogh, 2008], com o objetivo de explorar a representação multirresolução. Este esquema permite que o usuário obtenha *motifs* de diferentes resoluções. Essa propriedade permite o desenvolvimento de aplicações de visualização e navegação na estrutura hierárquica dos *motifs*. *MrMotif* é escalável e linear, devido a utilização de tabelas *hash* para o contagem de coincidências.

O pseudocódigo do *MrMotif* é apresentado no Algoritmo 3.5. O algoritmo é relativamente simples. Uma iteração inicial é usada para percorrer todas as séries temporais do conjunto de dados. Todas as séries são convertidas a uma palavra iSAX de uma resolução particular (linha 7). Note que a maioria de todas as conversões seguem um processo similar, e por isso cada conversão é executada em um só passo. Logo, se o *cluster* estiver presente na tabela *hash* $count_g$, o contador da estrutura é incrementado na localização da sequencia armazenada (linhas 9-12). De outro modo, o contador é inicializado em um (linha 14). Finalmente, os *TopK-Motifs* para cada resolução são imprimidos (linha 3).

Algoritmo 3.5: Algoritmo *MrMotif* para a procura aproximada do *TopK-Motifs*

```
1  algorithm MrMotif(S)
2    count_g := MrMotifInit(S)
3    print_motifs(count_g)
4  end.
5  procedure [count_g] = MrMotifInit(S)
6    foreach s_i in S do
7      W = iSAX(s_i, g_min, ..., g_max, word_size)
8      tree.Update(W)
9      foreach w_g in W do
10       if w_g is in count_g then
11         c_g := count_g.get(w_g)
12         count_g.Update(w_g, c_g + 1)
13       else
14         count_g.Update(w_g, 1)
15       end if
16     end for
17   end for
18 end.
```

3.7 Considerações Finais

Neste capítulo foi apresentada uma visão geral sobre mineração de séries temporais, destacando a consulta por similaridade e descoberta de *motifs*, que são tópicos de interesse para este trabalho. Apesar das recentes pesquisas em algoritmos exatos, algoritmos aproximados podem ser a melhor opção em muitos domínios de aplicação devido a sua eficiência em tempo e espaço. Nesses domínios, o *trade-off* entre tempo de execução e precisão da solução claramente se inclina para a primeira. Este fato é importante no embasamento da proposta deste trabalho.

Além disso, foram discutidas técnicas de representação e medidas de similaridade em séries temporais, que têm como objetivo minimizar o ruído e os efeitos da alta dimensionalidade em aplicações de mineração de séries temporais. Isto é importante, pois permite levar em consideração a natureza dos dados nos métodos de mineração.

CAPÍTULO
4

GPGPU em CUDA

4.1 Considerações Iniciais

As GPUs (*Graphical Processing Units* ou Unidades de Processamento Gráfico) estão se tornando onipresentes em computação de uso geral, em especial para aplicações de alto desempenho. Uma das razões que fundamenta esta evolução é que como as GPUs foram otimizadas para a renderização de gráficos em tempo real, estas também são idôneas para operações de computação intensiva e altamente paralela. Esse campo é conhecido como GPGPU (*General-Purpose computations on the GPU*).

Devido ao crescimento acelerado do poder de processamento das GPUs se comparado às CPUs, se evidencia, desta forma, que o poder de computação da primeira está crescendo a uma taxa maior do que nas CPUs. Essa evolução em computação de alto desempenho usando as GPUs gerou o desenvolvimento de aplicações não só orientadas para processamento gráfico, mas também em áreas que apresentam grandes volumes de dados para serem processados, principalmente aqueles voltados para a pesquisa científica, tais como computação científica, bioinformática, banco de dados, mineração de dados, computação distribuída [Owens J. e Purcell, 2007].

No esquema do GPGPU, a GPU é um coprocessador para a CPU. Como tal, a CPU é usada para o gerenciamento do disco e da memória, ou seja, é responsável por lidar com a leitura/escrita no disco, assim como pelas transferências de dados entre a memória da GPU e da CPU. O processamento das GPUs está baseado na arquitetura SIMD (*Single Instruction, Multiple-Data*), isto é, uma arquitetura de processamento paralelo em que vários conjuntos de dados podem ser processados simultaneamente pelo mesmo conjunto de instruções, de como que esses conjuntos de dados são executados em unidades de processamento distintas.

Por sua vez, as CPUs tradicionalmente possuem arquitetura SISD (*Single Instruction Single-Data*), sendo, pois baseadas em uma arquitetura de processamento sequencial. No entanto, ao mesmo tempo em que as GPUs mostraram ter um grande poder computacional, uma arquitetura de múltiplos núcleos impõe restrições no projeto dos algoritmos, assim como desafios para alcançar os picos de desempenho

mais altos.

Existem várias plataformas de programação GPGPU. Entre elas se destacam as seguintes: CUDA [Nickolls et al., 2008] e OpenCL [Stone et al., 2010]. A principal diferença entre as plataformas é que os programas CUDA são executados somente em placas de vídeo NVIDIA, enquanto nos programas OpenCL a execução é feita em plataformas heterogêneas. A distribuição de *kits* de desenvolvimento de software, como a NVIDIA-CUDA SDK, tem incentivado seu uso como uma forma de liberar a CPU de domínios de aplicação, que precisam de picos de desempenho mais altos. A arquitetura NVIDIA-CUDA foi escolhida para este trabalho por ser, no momento da escrita do texto, a plataforma dominante para computação de alto desempenho.

A plataforma CUDA proporciona a oportunidade de reduzir significativamente o tempo de execução de tarefas de recuperação e mineração de dados, pois geralmente apresenta seu hardware disponível e acessível em termos de valores, sem precisar de soluções de hardware mais complexas, como os servidores de aplicações de alto desempenho com configurações de múltiplas CPUs.

Em seguida, são apresentados alguns mecanismos de funcionamento da plataforma CUDA. Na Seção 4.2 são apresentados aspectos relacionados à plataforma de programação CUDA, como a arquitetura, o modelo de programação e o modelo de memória, elaboradas em [Kirk e Hwu, 2010]. Na Seção 4.3 são apresentados algumas restrições de implementação em CUDA. Na Seção 4.4 é apresentado um exemplo em CUDA em que se explora a capacidade *multi-thread* da plataforma. Na Seção 4.5 são discutidas algumas recentes aplicações em operações de consulta em banco de dados e tarefas de mineração de dados implementados em GPUs.

4.2 Arquitetura CUDA

A arquitetura de uma GPU NVIDIA típica, com suporte de tecnologia CUDA, é organizada em uma matriz de *Streaming Processors* (SPs) altamente segmentados, e que formam, assim, o conjunto de *Streaming Multiprocessors* (SMs). Todos os SM têm um número de *Streaming Processors* (SPs) que compartilham o controle lógico e a *cache* das instruções. As GPUs atualmente vêm com até 6 gigabytes de (GDDR)DRAM, conhecida como memória global. Por exemplo, a GeForce GTX 470 (G470) tem 1.280 MB de memória global e tem 448 SPs (14 SMs, cada um com 32 SPs). O chip G470 suporta até 1.024 *threads* por SM, que resume a cerca de 14.000 *threads* para este chip. Note que as CPUs Intel tradicionais suportam 2, 4 ou até 8 *threads*, dependendo do modelo da máquina.

A arquitetura CUDA de uma NVIDIA GeForce GTX 8800 é ilustrada na Figura 4.1. Ela consiste de 16 *Streaming Multiprocessors* (SMs), contendo 8 cores ou *Streaming Processors* (SPs). Os 8 cores executam as instruções das *threads* em SIMD, executando um $warp$ (um grupo de 32 *threads*, a unidade mínima de escalonamento) a cada 4 ciclos. Isso permite que cada multiprocessador possua apenas uma unidade de instruções, que envia a instrução corrente para todos os cores.

A NVIDIA oferece soluções para o GPGPU há muito tempo. A nova família de soluções da NVIDIA, pensada para aplicações em supercomputação, foi apresentada em 2010. Esta arquitetura é conhecida como $Fermi$. Baseando-se nesta nova arquitetura, a NVIDIA também apresentou sua plataforma para supercomputação chamada *Tesla* [Lindholm et al., 2008]. Essa arquitetura apresenta recursos como memórias com ECC e *double point precision*, o que é ideal para aplicações que exigem grande precisão, como por exemplo o *crast tests* ou túnel de vento na indústria automobilística.

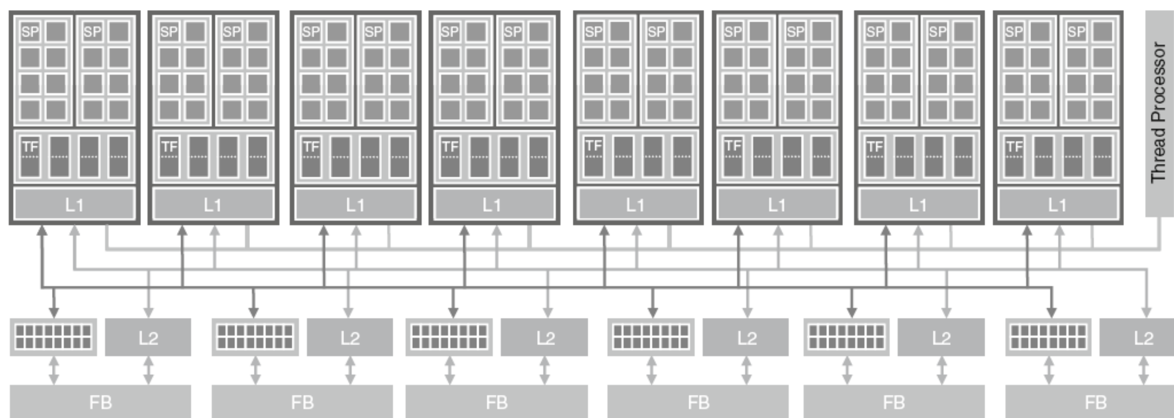


Figura 4.1: A arquitetura CUDA mostra um conjunto de processadores de unidades programáveis de uma NVIDIA GeForce GTX 8800. Figura adaptada de [Kirk e Hwu, 2010].

4.2.1 Modelo de Programação em CUDA

Um programa CUDA consiste em uma ou mais fases que são executadas, seja na CPU ou na GPU. As fases que apresentam pouco ou nenhum paralelismo de dados ([1]) são implementadas no ambiente de código da CPU, chamado *host*. Por outro lado, as fases que apresentam grande quantidade de paralelismo de dados são implementadas no ambiente de código na GPU, chamado *device*. Um programa CUDA é um código fonte que abrange tanto o código no ambiente *host* como no *device*.

O compilador da NVIDIA para CUDA, chamado NVCC, separa os dois ambientes de código durante o processo de compilação. O código *host* é código em C/C++, que será executado como um processo na CPU e também compilado ao se usar um compilador de C/C++ padrão. O código *device* é escrito ao se utilizar um compilador de C estendido com palavras-chave para se rotularem as funções e estruturas de dados associadas. O código *device* é compilado pelo NVCC e executado na GPU. Mais especificamente, os qualificadores usados para definir o tipo de função são:

- `__device__` função executada no *device* e invocada a partir de um ambiente *device*.
- `__glogal__` declara a função sendo um *kernel* e esta é executada no modo *device*, porém invocada somente a partir de um ambiente *host*.
- `__host__` função executada no *host* e invocada somente pelo *host*. Ao declarar uma função somente com o qualificador `__host__` equivale a não usar nenhum qualificador, pois, em ambos casos, a função será compilada somente para o *host*. Porém, se a função for usada junto como o qualificador `__device__`, a mesma será compilada para ambos modos, *host* e *device*.

Um *kernel* especifica o código a ser executado para todas as *threads* durante a fase paralela, e normalmente gera um grande número de *threads* para explorar o paralelismo de dados. É interessante notar que as *threads* CUDA são processos mais leves do que as *threads* da CPU. Em contraste com as *threads* da CPU, que normalmente requerem milhares de ciclos de relógio para serem geradas, os

[1] Paralelismo de dados: refere-se à propriedade do programa através da qual as operações aritméticas podem ser realizadas com segurança de forma simultânea sobre estruturas de dados.

programadores CUDA podem assumir que essas *threads* tomam poucos ciclos para serem geradas em razão ao suporte eficiente em hardware.

A execução de um programa em CUDA se inicia com a execução do $host$. Quando uma função $kernel$ é chamada ou lançada, a execução muda para o modo $device$, em que um grande número de *threads* é gerado para se beneficiar da arquitetura paralela. Todas as *threads* geradas por um $kernel$, durante uma chamada, são lançadas coletivamente por uma $grid$. O tamanho da $grid$ é especificado durante a chamada a uma função kernel usando a nomenclatura ilustrado na Figura 4.2. Por exemplo, nessa figura apresentasse o modo de execução de duas $grids$ de *threads*. No momento em que todas as *threads* de um $kernel$ completam a sua execução, a $grid$ correspondente termina, e a execução continua no *host* até que outro $kernel$ seja invocado.

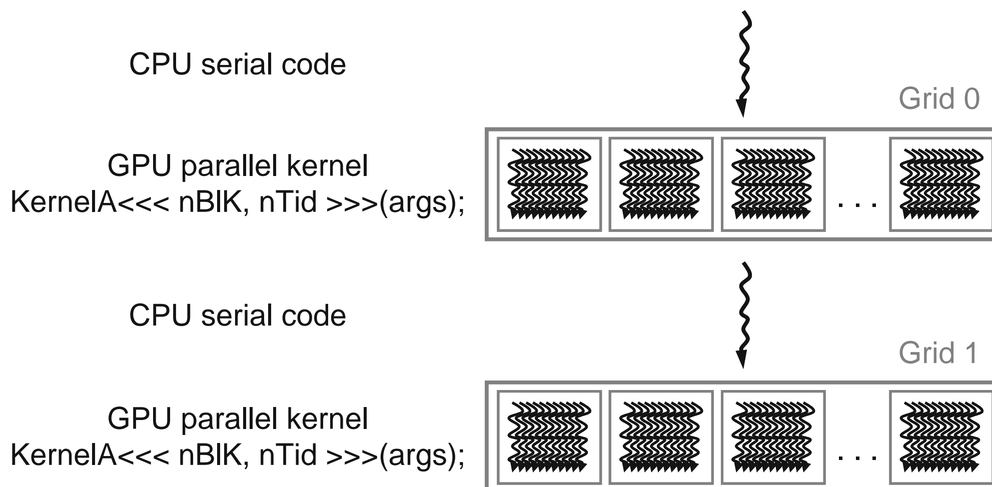


Figura 4.2: Execução de um programa em CUDA. Figura adaptada de [Kirk e Hwu, 2010].

As *threads* em uma $grid$ são organizadas por uma hierarquia de dois ou três níveis, dependendo da implementação. Por exemplo, na Figura 4.2, o $KernelA$ é lançado e gera o $Grid$ 0, esta $grid$ é organizada como uma matriz 2×2 de 4 blocos.

Em geral, cada bloco dispõe de duas ou três coordenadas dimensionais dadas por palavras chave `blockIdx.x`, `blockIdx.y` e `blockIdx.z`. Todos os blocos devem ter o mesmo número de *threads* organizados de maneira uniforme. No nível superior, cada $grid$ é composta de vários blocos de *threads*, sendo que cada bloco tem o mesmo número de *threads* e, por sua vez, a $grid$ é organizado como uma matriz tridimensional de *threads*.

À título de verificação, um pequeno número de *threads* é ilustrado na Figura 4.3 em que, cada bloco de *threads* é organizado em uma matriz tridimensional de $4 \times 2 \times 2$. Obtendo, desta forma. uma $grid$ de $4 \times 16 = 64$ *threads*. Este esquema é, obviamente, um exemplo simplificado, pois na prática é geralmente muito mais complexo. Por exemplo, no chip G470, um bloco de *threads* tem um tamanho total de até 1024 *threads*.

Para sincronizar *threads* no mesmo bloco, a plataforma de programação CUDA permite coordenar suas atividades utilizando a função de sincronização `__syncthreads ()`. Tal condição garante que todos os *threads* em um bloco pessam completar uma fase de execução do $kernel$, antes mesmo de se moverem para a próxima fase.

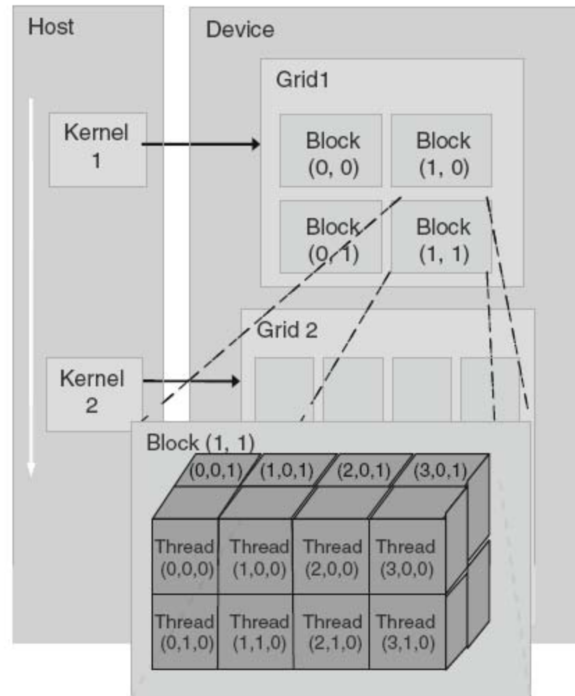


Figura 4.3: Organização dos *threads* em CUDA. Figura adaptada de [Kirk e Hwu, 2010].

4.2.2 Modelo de Memória

Em CUDA, os modos de programação *host* e *device* apresentam espaços de memória separados. Isso mostra que em CUDA os dispositivos $host$ (CPU) e $device$ (GPU) são, geralmente, elementos de programação que têm suas próprias memórias. Portanto, a fim de executar um $kernel$, o programador precisa alocar a memória na GPU e transferir dados pertinentes da memória da CPU para a memória da GPU. Da mesma forma, depois da execução do $kernel$, o programador precisa transferir os dados do resultado da memória da GPU de volta para a memória da CPU e, assim, liberar a memória na GPU que não é mais necessária. O sistema de execução CUDA proporciona as funções necessárias para realizar estas atividades.

CUDA suporta vários tipos de memória que podem ser utilizados por programadores, como o intuito de alcançar altas velocidades de execução em seus $kernels$. A Figura 4.4 ilustra os modelos de memórias em um dispositivo CUDA. Na parte inferior da figura, observa-se a memória global e a memória constante. Esses tipos de memória podem ser escritos e lidos pelo *host* chamando funções da API *Application Programming Interface* (ou Interface de Programação de Aplicativos). A memória global é maior e também a mais lenta, enquanto a memória constante é pequena (64 KB), mas, contudo é mais rápida e suporta, basicamente, apenas a só leitura, quando todas as *threads* acessam simultaneamente a mesma localização.

Os registros e a memória compartilhada ilustrado nessa na Figura 4.4 são memórias *on-chip*. As variáveis que residem nesses tipos de memória podem ser acessadas em alta velocidade de maneira altamente paralela. A função $kernel$ geralmente usa registros que contêm as variáveis acessadas frequentemente, sendo estas particulares para cada *thread*. Ao passo que a memória compartilhada é alocada para blocos de *thread*, todas as *threads* em um bloco podem acessar as variáveis na memória

compartilhada. Nota-se que a memória compartilhada é um meio eficiente para que as *threads* cooperem por meio de seus dados de entrada e pelos resultados intermediários.

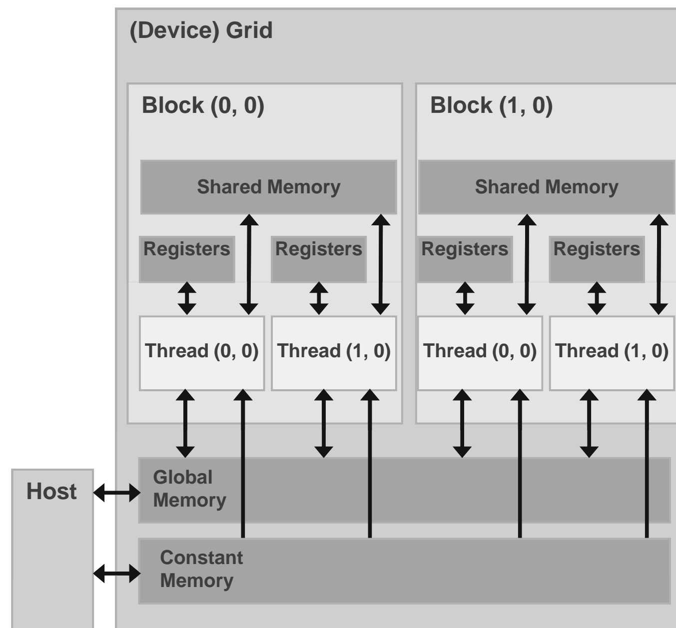


Figura 4.4: Modelo de memória em CUDA. Figura adaptada de [Kirk e Hwu, 2010].

Por esse modo, se uma declaração de variável é precedida pela palavra chave `__shared__`, isso indica que uma variável é compartilhada em CUDA. Em contrapartida, se uma declaração de variável é precedida pela palavra chave `__constant__`, ela declara uma variável constante em CUDA. Uma variável cuja declaração é precedida apenas pela palavra chave `__device__` É, pois, uma variável global e assim será colocada na memória global.

4.3 Restrições de implementação

Para desenvolver soluções algorítmicas em CUDA precisa-se considerar algumas restrições do modelo de programação de CUDA. Por exemplo, não pode haver recursão, variáveis estáticas, funções com número de argumentos variáveis e ponteiros para funções, elementos de *structs* e *unions* devem pertencer ao mesmo espaço de endereçamento. Assim, a maioria de soluções desenvolvidas para CPU precisam ser reprojetadas considerando desenhos de algoritmos e estrutura de dados simples para atingir altos níveis de desempenho.

Em alguns casos, a diferença em termos de desempenho entre um código em CPU e uma implementação massivamente paralela em GPU é insignificante, especialmente quando se trabalha com estruturas de dados complexas. Isto porque as GPUs não foram projetadas para se destacarem em acesso *multi-thread* a estruturas de dados complexas, tais como árvores e tabelas *hash*. Por essa razão, há pouca motivação para se desenvolver estruturas de dados complexas na GPU.

Num outro cenário, em que o *pipeline* de um programa longo envolve um ou dois estágios, ao passo que a GPU não usufrui de uma vantagem de desempenho sobre as implementações em CPU. Nessas situações, têm-se as três opções seguintes:

- Executar todas as etapas do *pipeline* na GPU;
- Executar todas as etapas do *pipeline* na CPU;
- Executar alguns passos do *pipeline* na GPU e outros na CPU.

A última opção parece ser a melhor escolha, pois combina o melhor dos dois mundos. No entanto, isso implica que será preciso sincronizar a CPU e a GPU em qualquer ponto da aplicação pelo qual se deseja mover o cálculo da GPU para a CPU, ou vice-versa. Essa sincronização e a subsequente transferência de dados entre o $host$ e o $device$, pode limitar muitas das vantagens de desempenho da GPU. Em tal situação, pode valer a pena realizar todas as fases da computação na GPU, mesmo se a GPU não for ideal para alguns passos do algoritmo.

Nesse sentido, uma implementação em GPU com a maioria das etapas do *pipeline* do programa pode minimizar a transferência de dados. Em tal cenário, é possível que o desempenho geral de uma implementação apenas em GPU seja superior à abordagem híbrida CPU/GPU, mesmo que algumas etapas do *pipeline* em GPU não sejam mais rápidas do que na CPU.

4.4 Exemplo em CUDA

Problema do Par mais Próximo implementado em CUDA

O algoritmo de força bruta para o problema do par mais próximo (*Closest Pair*) é uma questão bem conhecida e estudada em computação. Esse problema na área de mineração de séries temporais também é conhecido como a procura do *Pair-Motif*. Como foi apresentado na Seção 3.6.1, o algoritmo de força bruta tem um custo computacional quadrático.

Nesta seção, apresenta-se uma implementação massivamente paralela do algoritmo de força bruta, em que se utiliza uma das vantagens mais destacadas do modelo de programão em CUDA, ou seja, a geração automática dos identificadores para os *threads*. Isto é realmente útil para quando o problema é altamente paralelizável e divisível em vários níveis, como por exemplo, em blocos de *threads* que podem ser bidimensionais ou até tridimensionais.

Algoritmo 4.1: Algoritmo CUDA-ClosestPair (versão 1)

1 **algorithm** $CUDA{-}ClosestPair(S)$
2   **deviceMem** $bsf := \infty$
3   **deviceMem** $S'[][] := S[][]$
4     $blocks := < 65535, 65535 >$
5   *threads* $:= < 1, 1 >$
6     $[L_1, L_2] := ClosestPair{-}Kernel<<<blocks, threads>>>(S', bsf)$
7   Copia as variaveis L_1 e L_2 à mémoria principal da CPU
8   Liberar a mémoria associada as variaveis bsf e S' na GPU
9 **end**.

O Algoritmo 4.1 apresenta o pseudocódigo associado à implementação do Par mais Próximo (*Closest Pair*) em CUDA. Nota-se que para executar um processo em CPU, chama-se só ao procedimento associado, enquanto que para o algoritmo em CUDA, cria-se uma função em CPU que realiza o seguinte: (1) aloca-se a memória associada ao processo no *host* e no *device*; (2) configura-se os blocos de *threads*,

e, neste caso um *thread* por cada bloco (linha 4); (3) envia-se os dados para ser processado na GPU usando a função $kernel$ (linha 5); (4) após a execução do $kernel$ o algoritmo copia os resultados para a CPU e limpa a memória alocada na memória da GPU (linhas 6 e 7).

O Algoritmo 4.3 apresenta a função $kernel$ para o problema da busca do par mais próximo. Verifica-se que a função $kernel$ em CUDA é bem simples, pois os dois *loops* do Algoritmo em CPU 4.2 (linhas 3 e 4) são agora substituídos por blocos de *threads* (linhas 2 e 3). Cada *thread* do bloco corresponde a uma iteração no algoritmo em CPU. As variáveis i e j são, pois, substituídas por identificadores dos blocos no eixo x e no eixo y. Assim, em vez de usar os valores i e j no *loop* incremental de dois níveis, o programa em CUDA gera todos os valores para as variáveis i e j. Mediante uma rápida comparação entre os algoritmos 4.2 e 4.3 revela essa importante característica de CUDA.

Algoritmo 4.2: ***Closest Pair*** **na CPU**

```
1  algorithm [L1, L2] = ClosestPair(S)
2    bsf := ∞;
3    for i := 1 to N do
4      for j := i + 1 to N do
5        dist := distance(s_i, s_j)
6        if dist < bsf then
7          bsf := dist
8          L1 := i, L2 := j
9        end if
10     end for
11   end for
12 end.
```

Algoritmo 4.3: ***Closest Pair*** **na GPU**

```
1  kernel [L1, L2] = ClosestPair−Kernel(S, bsf)
2    i := blockIdx.x
3    j := blockIdx.y
4    if i < j then
5      dist := distance(s_i, s_j)
6      prev_bsf := atomicMin(bsf, dist)
7      if dist < prev_bsf then
8        atomicExch(L1, i)
9        atomicExch(L2, j)
10     end if
11   end if
12 end.
```

No entanto, existem algumas limitações com essa implementação. O bloco de *threads* é limitado em tamanho. Ele só pode ter até $65535 * 65535$ blocos por $grid$ e até 1024 *threads* por bloco. Para superar esta dificuldade, podemos dividir o problema em várias partições de tamanho $twidth = 32$, ou seja uma combinação de *threads* e blocos com $32 * 32$ *threads* por bloco. Esta solução é mais geral, dado que o problema é subdividido em pequenas partições. Desse modo, O número de blocos é de $N/twidth * N/twidth$ e o número de *threads* por bloco é de $twidth * twidth$, obtendo, desta forma, uma $grid$ de $N * N$ *threads*. Desse modo, cada *thread* da $grid$ de *threads* corresponde a uma iteração no algoritmo em CPU. O algoritmo aperfeiçoado de força bruta implementado em CUDA é descrito no Algoritmo 4.4.

Para compreender o código nas linhas 11 e 12, deve-se notar que, usando uma combinação de *threads* e blocos, pode-se obter um único índice ao utilizar o produto do índice do bloco (blockIdx) com o número de *threads* em cada bloco (blockDim) e, por fim, adicionar o índice do *thread* dentro do bloco (threadIdx). Isso é idêntico ao método utilizado para linearizar o índice de imagem bidimensional ($offset = x + y * DIM$). Indexar os dados em uma matriz linear usando ao método anterior, é, na verdade, bastante intuitivo, uma vez que se pode pensar nele como uma coleção de blocos de *threads*. Na Figura 4.5 ilustra-se a disposição bidimensional de blocos e *threads* que representam o esquema de partição utilizado no Algoritmo 4.4.

Algoritmo 4.4: Algoritmo CUDA-ClosestPair (versão 2)

```
1  algorithm CUDA−ClosestPair(S)
2      deviceMem S'[][] := S[][]
3      deviceMem bsf = ∞
4      threads := < twidth, twidth > //twidth = 32
5      blocks := < N/twidth, N/twidth >
6      [L1, L2] := ClosestPair−Kernel<<<blocks, threads>>>(S', bsf)
7      Copia as variaveis L1 e L2 à memória principal da CPU
8      Liberar a mémoria associada as variaveis bsf e S' na GPU
9  end.
10 kernel [L1, L2] = ClosestPair−Kernel(S, bsf)
11     i := blockIdx.x ∗ blockDim.x + threadIdx.x
12     j := blockIdx.y ∗ blockDim.y + threadIdx.y
13     if i < j then
14         dist := distance(Si, Sj)
15         prev_bsf := atomicMin(bsf, dist)
16         if dist < prev_bsf then
17             atomicExch(L1, i)
18             atomicExch(L2, j)
19         end if
20     end if
21 end.
```

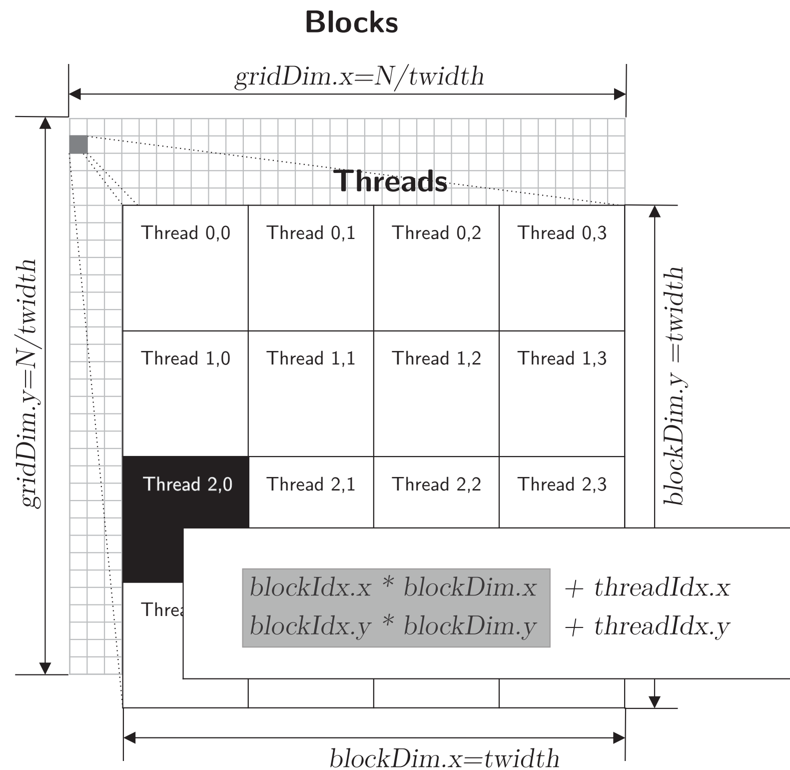


Figura 4.5: Disposição bidimensional de uma coleção de blocos e *threads* utilizado no Algoritmo 4.4.

4.5 Aplicações

Nesta seção, são discutidas as pesquisas relacionadas às aplicações de propósito geral usando GPUs, com foco especial nas operações de processamento de consultas em banco de dados e também nas tarefas de mineração de dados.

4.5.1 Operações de processamento de Consultas em Banco de Dados utilizando GPUs

A capacidade computacional das GPUs tem sido exploradas no processamento de consultas e operações relacionadas. Por exemplo, em [Govindaraju et al., 2006, Govindaraju et al., 2004] demonstraram que blocos de construção importantes para o processamento de consultas em banco de dados, como a ordenação, a seleção conjuntiva, a agregação e as consultas sublineares podem ser acelerados significativamente com o uso de GPUs.

Alem disso, em [Lieberman et al., 2008] abordam o tema de junção por similaridade no espaço n-dimensional para pares de objetos a partir de dois diferentes conjuntos R e S, cumprindo dessa forma certo predicado de junção. O predicado de junção mais comum é o ϵ-$join$ que determina todos os pares de objetos em uma distância menor que um $threshold$ ϵ predefinido.

A capacidade computacional das GPUs também tem sido explorada para melhorar consultas kNN. Para exemplificar, em [Garcia et al., 2008] descreve-se uma solução de consultas kNN usando a abordagem de força bruta, mas utilizando-se de uma implementação altamente paralela em CUDA. Numa abordagem similar, em [Liang et al., 2010] apresentou o método CUKNN. O CUKNN resulta em um aumento de velocidade média de 46 vezes, se comparado com o método sequencial de busca kNN, baseado no *quick sort*.

Num outro trabalho, consultas kNN e AllkNN baseadas em kd-trees foram propostas em [Zhou et al., 2008]. Devido às restrições do modelo de programação de CUDA, utilizaram-se *arrays* com a mesma abordagem do $heapsort$ para representar as árvores binárias. Essa restrição resultou na construção da árvore em CPU para logo ser representada num *array* em GPU, o que é limitação no desempenho, pois para poder minimizar a custo de transferência de dados se prefere implementações em CUDA com a maioria das etapas do *pipeline* do programa em GPU. Outra restrição dessa implementação é que só trabalha para dados em dois e três dimensões.

Estruturas de dados espaciais e métricos podem ser utilizadas para acelerar as consultas kNN, mas a arquitetura CUDA não foi projetada para processar estruturas de dados complexas. É por essa razão que há pouca motivação para se desenvolver estruturas de dados para buscas kNN em GPU. Por isso, a lacuna relacionada a métodos de indexação para buscas kNN implementados em GPU é uma das motivações deste trabalho.

4.5.2 Tarefas de Mineração de Dados utilizando GPUs

Devido ao crescente poder das GPUs, as soluções GPGPU para mineração de dados de alto desempenho podem ser adequadas. Em problemas de grande escala, arquiteturas sequenciais (CPUs) impõem limitações de desempenho para oferecer soluções em tempo real. Portanto, recorrer às soluções baseadas em GPU é uma maneira para se superar tais limitações de desempenho.

Recentemente, a capacidade computacional de múltiplos núcleos das GPUs tem sido utilizada para acelerar muitos problemas de computação, incluindo alguns no domínio do agrupamento de grandes volumes de dados. A mineração de dados paralela em GPU foi avaliada em [Wu et al., 2009, Che et al., 2008]. Essas abordagens se utiliza o algoritmo de $k - means$ para agrupar grandes conjuntos de dados, que na prática, são limitados pela memória local da GPU. Pesquisas anteriores demonstraram que, usando uma GPU para agrupamento de milhares de documentos, pode-se ter uma melhoria de até cinco vezes na velocidade dos algoritmos [Charles et al., 2008]. Esse fato demonstra os benefícios em se utilizar de arquiteturas GPU para problemas altamente paralelizáveis.

Numa abordagem mais recente, em [Böhm et al., 2009] se utilizaram GPUs para paralelizar a junção por similaridade e, consequentemente, se valeu deste operador para acelerar o algoritmo de agrupamento k-means, alcançando melhorias importantes no desempenho quando comparado com algoritmos sequenciais.

Na área de bioinformática, o método MEME [Bailey e Elkan, 1995] é um algoritmo iterativo comumente utilizado para encontrar *motifs* em conjunto de sequências de proteínas. Infelizmente, a sua complexidade é de $O(N^2L^2)$, em que M é o número de sequências de entrada e L é o comprimento de cada sequência, o que torna esse algoritmo impraticável para conjuntos de dados muito grandes. Com a crescente quantidade de dados, há uma grande necessidade de algoritmos eficientes. CUDA-MEME, desenvolvido por Liu et.al [Liu et al., 2010], é uma implementação altamente paralela do algoritmo MEME. O CUDA-MEME resulta em um aumento de velocidade média de 20 vezes para o modelo OOPS e de 16 para o modelo ZOOPS, em comparação ao algoritmo MEME sequencial. O tempo de execução do CUDA-MEME sobre uma GPU é comparável à implementação paralela ParaMEME executada em uma máquina de 16 CPUs. No entanto, como o custo da GPU utilizada é menor do que o custo da máquina paralela, o CUDA-MEME não é apenas a ocorrência de uma paralelização eficiente do algoritmo MEME, mas também uma solução economicamente mais viável.

4.6 Considerações Finais

Neste capítulo foi discutido o uso da GPU para propósito geral, campo conhecido como GPGPU. Nas GPUs tradicionais, a computação de propósito geral é efetuada quando são utilizadas as APIs gráficas. Contudo, com o surgimento das APIs, tais como CUDA ou OPENCL, o desenvolvimento de GPGPU se tornou mais simples. Nesse sentido, a API CUDA é a mais evoluída e possui características como a flexibilidade de programação, o uso da linguagem C e maior largura de banda das memórias tornam a API muito atraente para computação de alto desempenho.

Além disso, foram apresentadas algumas aplicações nas áreas de banco de dados e mineração de dados, destacando a consulta por similaridade e a descoberta de *motifs*, sendo estes os tópicos de interesse para este trabalho. Apesar das recentes implementações desses algoritmos, encontrou-se uma lacuna relacionada aos algoritmos aproximados, sendo os mesmos aplicados aos problemas de mineração de dados, em que soluções simples e gerais são consideradas impraticáveis em um contexto real, devido ao seu alto custo computacional. Um exemplo disso são os problemas tanto da busca a todos os k-vizinhos mais próximos, como o problema da procura dos K padrões mais frequentes. Assim, neste trabalho são propostas soluções aproximadas implementadas em CUDA para esses problemas, como descrito no Capítulo 5.

CAPÍTULO

5

Soluções Aproximadas de Busca por Similaridade e Descoberta de *Motifs* usando GPGPU

5.1 Considerações Iniciais

Este trabalho de mestrado tem como principal objetivo investigar e desenvolver soluções baseadas em busca por similaridade aproximada por meio da computação GPGPU, como forma de reduzir o custo computacional de tarefas de mineração de dados que utilizam o kNN como procedimento mais importante. Soluções aproximadas para a busca por similaridade devem ter um forte modelo teórico para assim garantirem a precisão e a escalabilidade. Além disso, para aproveitar os recursos do paradigma de programação GPGPU, a solução deve apresentar algoritmos e estrutura de dados simples. Desse modo, uma vez determinada uma solução adequada para buscas por similaridade, e sendo esta estendida e adaptada para explorar os recursos da técnica GPGPU, especificamente a plataforma de programação CUDA, se melhora o desempenho dos algoritmos para a identificação de *motifs* visando a escalabilidade dos algoritmos.

Neste capítulo é apresentado o método CUDA-LSH, projetado neste trabalho para fornecer buscas por similaridade de alto desempenho, especificamente a busca paralela AllkNN (Seção 5.2). A proposta é baseada nos esquemas de indexação *Locality Sensitive Hashing* (LSH) [Datar et al., 2004] e no HashFile [Zhang et al., 2011]. O LSH é uma solução aproximada de busca por similaridade que permite consultas de custo sublinear para dados em altas dimensões. O HashFile é uma extensão do esquema de indexação LSH que apresenta o melhor equilíbrio entre desempenho e uso de espaço. Além disso, os esquemas de indexação LSH podem ser adaptados para trabalhar com estruturas de dados simples, o que é vantajoso, pois a arquitetura CUDA impõe algumas restrições de implementação no modelo de programação. Por essa razão, muitas das outras soluções, tanto as exatas como as aproximadas, que apresentam estruturas de dados complexas não foram consideradas como possíveis soluções para as buscas de alto desempenho

em GPU.

A partir do esquema de indexação CUDA-LSH para a busca por similaridade de alto desempenho, os algoritmos de descoberta de *motifs* (Seção 6.3.3) apresentaram um melhor desempenho usando implementações eficientes e paralelas de buscas AllkNN. Portanto, uma vez definidos os principais procedimentos do método CUDA-LSH, foi proposto o algoritmo paralelo de descoberta de *motifs*, *CUDA-TopkMotifs*, baseado na definição dos K padrões mais frequentes. Essa implementação massivamente paralela é descrita neste capítulo (Seção 5.3).

Todas as implementações dos algoritmos de busca AllkNN e descoberta de *motifs* foram elaboradas para GPUs com suporte para a arquitetura CUDA. Em particular, paralelizar e otimizar tais algoritmos para GPUs não é uma tarefa trivial, a maioria das implementações precisa considerar algumas restrições de desenho para maximizar a utilização dos núcleos da GPU e também minimizar o fluxo de dados entre a CPU e a GPU. Assim, algumas técnicas de otimização em GPU descritas no Capítulo 4 foram utilizadas para o desenvolvimento das funções $kernel$.

5.2 Visão Geral do método de indexação CUDA-LSH

Nesta seção, descreve-se a técnica de indexação CUDA-LSH, desenvolvida com o intuito de fornecer buscas de alto desempenho para consultas por similaridade aproximada em conjuntos de dados multidimensionais. A estratégia a seguir é de criar uma implementação massivamente paralela do esquema de indexação LSH considerando uma estrutura de índice multinível. A abordagem, como no esquema LSH, é baseada na ideia de projeção de objetos multidimensionais em espaços de busca mais simples, nas quais a intra-similaridade entre os objetos é preservada. No entanto, ao contrário do LSH sequencial, o CUDA-LSH é capaz de realizar as operações de construção e a consulta em paralelo.

Com base no esquema de indexação LSH [Datar et al., 2004] e HashFile [Zhang et al., 2011], as estruturas de dados e algoritmo de construção e consulta foram adaptadas e redefinidas para que possam se beneficiar dos recursos de computação para alto desempenho fornecidos pela GPU, como detalhado nas seções seguintes.

5.2.1 Organização da Estrutura

Como apresentado em [Datar et al., 2004], o LSH original compõe-se de L tabelas *hash*, em que, ao se utilizar um conjunto independentes de funções *hash*, cada tabela *hash* indexa o conjunto de dados inteiro, tal que objetos semelhantes são armazenados no mesmo *bucket*. Por outro lado, o HashFile [Zhang et al., 2011] utiliza a abordagem de projeção aleatória, a fim de mapear dados multidimensionais em valores reais, de modo que os dados são armazenados sequencialmente em ordem crescente ao seu valor *hash*. Desse modo, os esquemas de indexação LSH podem ser adaptados para trabalhar com estruturas de dados simples na GPU.

No entanto, o LSH expõe algumas desvantagens como os altos requisitos de espaço. Isso significa que esse esquema usa vários subíndices, com o objetivo de garantir resultados de alta qualidade. Além disso, o LSH tem uma dependência crítica sobre os parâmetros de domínio que determinam o número de funções e tabelas *hash*. Por outro lado, o HashFile contém apenas a dependência de um parâmetro, ou seja, o tamanho da janela de partição w, e a utilização de um único índice e, como consequência, ao

contrário do LSH, não precisa de vários subíndices.

Para resolver o problema de casos em que se faz necessário acessar a maioria dos pontos para responder consultas com altos níveis de qualidade, o HashFile se vale de uma estratégia inteligente para a partição de *buckets* densos. Essa estratégia melhora a partição do espaço usando melhores funções *hash* para os *buckets* densos, fazendo com que o custo de acesso aos dados também seja reduzido. Entretanto, tem-se a desvantagem de que se precisa utilizar estruturas de dados mais complexas, além de definir o tamanho da janela de partição w.

Para projetar o novo esquema da estrutura de dados para a implementação paralela em CUDA, não só é necessário considerar tais questões, mas também algumas restrições e desafios de implementação da plataforma de programação CUDA, a fim de atingir altos níveis de desempenho. Assim, considera-se uma estrutura de dados muito mais simples. A estrutura de dados é um *array* multinível, em que cada nível é projetado como um subíndice independente. Os dados são organizados a partir da estratégia de mapeamento de dados definida pelo HashFile, de forma que em cada nível os dados são armazenados sequencialmente em ordem crescente ao valor *hash*. A Figura 5.1 ilustra a estrutura lógica da estrutura de dados multinível do CUDA-LSH.

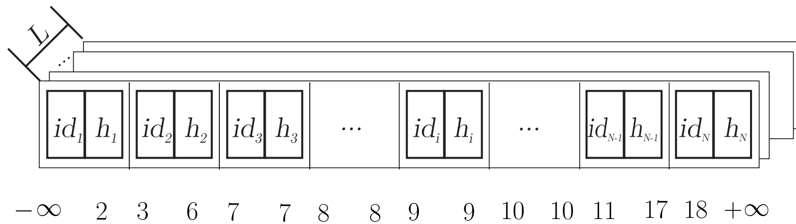


Figura 5.1: Estrutura lógica do índice CUDA-LSH.

Cada entrada $< id, h >$ em uma estrutura multinível é composta pelo identificador do objeto id_i; e pelo valor *hash* h_i deste objeto, que é obtido pela Equação 2.11 (Seção 2.5.3). Dessa forma, cada nível se associa a uma função *hash* $\mathcal{H}$. Essa estrutura pode ser representada como:

$$\textit{nível}\ [\textit{vetor}\ [1...N]\ \textit{de}\ < id_i, h_i >]$$

5.2.2 Construção do CUDA-LSH

As estruturas de dados utilizadas nos esquemas de indexação LSH foram adaptadas para trabalharem na memória da GPU. Deste modo, as estruturas de dados multinível são utilizadas como subíndices do método CUDA-LSH. Ao se utilizar dos vários níveis pode-se melhorar a qualidade dos resultados para consultas por similaridade aproximada. Por conseguinte, com o propósito de mapear objetos multidimensionais em valores unidimensionais, a função *hash* apresentada na Equação 2.11 é utilizada.

Nota-se que em razão da ausência de necessidade de utilizar a estratégia de partição do HashFile aplicado aos *buckets* densos, não é necessário definir o tamanho da janela de partição w. Portanto, a proposta do CUDA-LSH simplifica tanto as estruturas de dados quanto a dependência de parâmetros, ao utilizar apenas os parâmetros λ (parâmetro ajustável de erro) e L (número de níveis) como variáveis ajustáveis de precisão para consultas por similaridade aproximada.

Enquanto o método de indexação LSH processa todo o conjunto de dados sequencialmente, para indexar dados multidimensionais paralelamente deve-se considerar adaptações no algoritmo de construção. Desse modo, é preciso que o algoritmo de indexação processe cada elemento do conjunto de dados de forma independente. Assim, no esquema de indexação do CUDA-LSH, primeiro mapeia-se de forma paralela todos os objetos do conjunto de dados, depois ordena-se os elementos, de modo que os objetos são dispostos em ordem crescente do seu valor *hash*. A Figura 5.2 ilustra a disposição do índice CUDA-LSH antes e depois do processo de ordenação. Utiliza-se uma escala de cores para representar como os valores *hash* são distribuídos antes e depois do processo de ordenação dos dados.

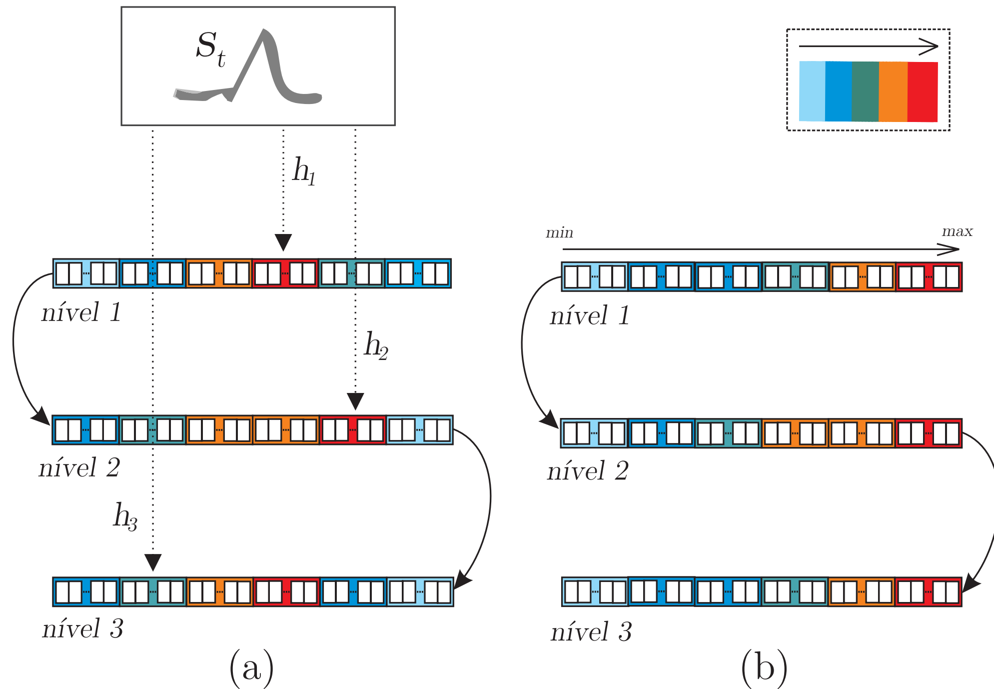


Figura 5.2: Estrutura do índice CUDA-LSH em suas etapas distintas de construção (a) Projeção de todos os objetos (b) Ordenação em ordem crescente.

O Algoritmo 5.1 apresenta o pseudocódigo do processo de construção do índice em CUDA-LSH. Note que para executar um processo em CPU, chama-se só o procedimento associado, enquanto que para o algoritmo em CUDA, cria-se uma função em CPU que realiza o seguinte:

1. Aloca-se memória associada ao processo na CPU e na GPU;
2. Inicializa-se as funções *hash* associadas à estrutura de dados (linha 2);
3. Configura-se os blocos de *threads*, neste caso $nThreads$ para os $(N/nThreads)$ blocos (linhas 4 e 5);
4. Envia-se os dados para serem processados na GPU usando a função $kernel$ (linha 6);
5. Após a execução da função $kernel$ o algoritmo ordena os dados em ordem crescente, de acordo com o seu valor *hash* (linha 7).

As etapas do processo de indexação paralela para o CUDA-LSH é ilustrada na Figura 5.3. Basicamente, essa figura ilustra as etapas do *pipeline* para o Algoritmo 5.1, em que se prepara a estrutura do índice para o processamento de consultas.

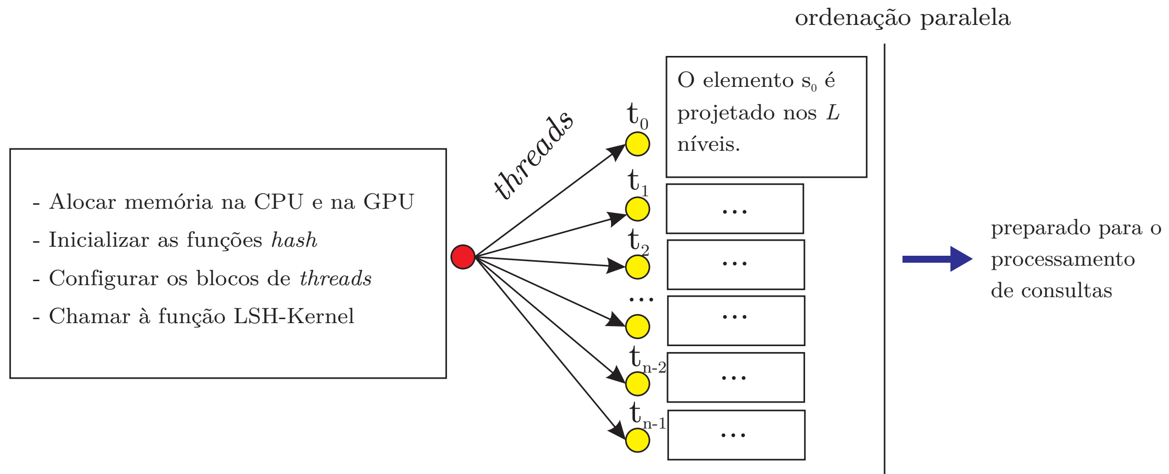


Figura 5.3: Etapas do processo de indexação paralela do CUDA-LSH.

Note que o algoritmo de construção utiliza uma das vantagens mais importantes do modelo de programação em CUDA, isto é, a geração automática dos identificadores para os *threads* (linha 11). Esses identificadores são associados, por sua vez, com os identificadores dos objetos (id_i), ou seja, cada objeto é processado paralelamente por um *thread*. O mesmo esquema de paralelização é utilizado para o problema de busca AllkNN como será monstrado na próxima seção.

Algoritmo 5.1: Construção do índice CUDA-LSH

```
1  algorithm CUDA−LSH(S)
2      deviceMem LSH := initHashFunctions() // inicialização das funções hash
3      deviceMem S'[][] := S[][]
4      threads := nTheads
5      blocks := N/nTheads
6      LSH−Kernel<<<blocks, threads>>>(S', LSH)
7      cuda_sort(LSH.keys, LSH.ids)
8  end.
9  kernel LSH−Kernel(S, LSH)
10     t := blockIdx.x ∗ blockDim.x + threadIdx.x // point id
11     for i := 1 to L do
12         hashvalue := LSH_i.hash(s_t)
13         LSH_i.keys[t] := hashvalue
14         LSH_i.ids[t] := t
15     end for
16 end.
```

5.2.3 Consultas kNN aproximada no CUDA-LSH

Para uma implementação massivamente paralela do algoritmo de busca AllkNN, precisa-se primeiro definir um algoritmo suficientemente simples de buscas kNN para ser utilizado como procedimento

principal do algoritmo AllkNN. Assim, a seguir apresenta-se o algoritmo de busca kNN para o método CUDA-LSH.

Para responder consultas por similaridade, uma vez que as estruturas de dados multinível foram criadas, as buscas são realizadas projetando os objetos de consulta no espaço de busca definido pela estrutura de dados de L níveis e as funções *hash* associadas. Desse modo, os objetos de consulta são mapeados em posições em que objetos similares têm alta probabilidade de serem localizados, reduzindo assim o número de cálculos de distância. As funções *hash* mapeiam objetos multidimensionais a um espaço unidimensional, em que uma relação de ordem é definida com base na proximidade entre objetos no espaço original. Uma vez que os objetos são armazenadas sequencialmente em ordem crescente ao valor *hash*, para cada objeto de consulta q e um parâmetro ajustável de erro λ, se o valor *hash* de q é h_q, então, de acordo com a relação apresentada na Equação 2.12 (Seção 2.5.3), apenas os elementos com valor *hash* entre $[h_q - \lambda, h_q + \lambda]$ precisam ser processados. Os demais elementos fora do intervalo não precisam ser processados. Nota-se que λ especifica o intervalo de consulta em vez do número de *buckets* vizinhos.

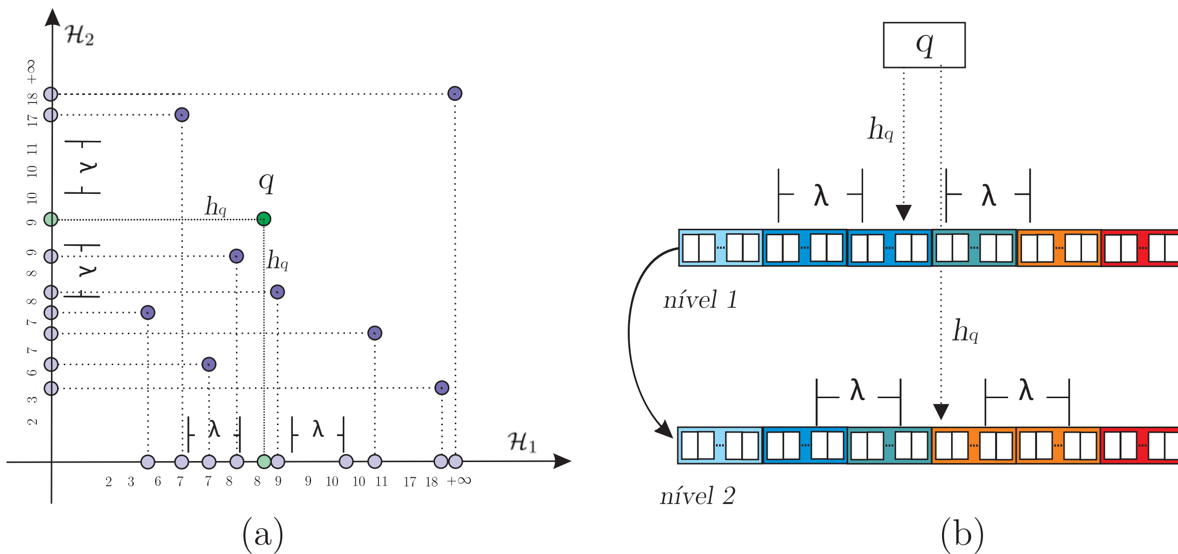


Figura 5.4: Representação das projeções de uma consulta por similaridade aproximada no CUDA-LSH em (a), em (b) verifica-se a sua estrutura lógica correspondente.

A Figura 5.4(a) apresenta as projeções de uma consulta por similaridade aproximada no CUDA-LSH, em que o objeto de consulta q é mapeado usando as funções *hash* ($\mathcal{H}_1$ e $\mathcal{H}_2$) nos dois níveis da estrutura. A sua estrutura lógica correspondente é apresentada na Figura 5.4(b). Nesse exemplo, as projeções geram partições diferentes no espaço de busca, em que cada função *hash* é utilizada para organizar todo o conjunto de dados, de modo que a proximidade dos objetos no espaço original seja mantida na projeção. Durante o processamento da consulta, para cada uma das projeções o objeto de consulta q é projetado para o valor *hash* h_q, e apenas os objetos dentro do intervalo *hash* $[h_q - \lambda, h_q + \lambda]$ precisam ser analisados. Desse modo cada nível contribui na precisão dos resultados ampliando o espaço de busca.

O pseudocódigo para responder consultas kNN é apresentado no Algoritmo 5.2. Para que se possa responder às consultas kNN, uma estrutura de dados $heap$ é utilizada para armazenar os k vizinhos mais próximos encontrados. A estrutura de dados atua como um *array*, logo, como um $heap$. Os primeiros

$k-1$ elementos visitados são anexados no final do *array*. Após adicionar o k-ésimo elemento no *array*, este último é convertido em um $heap$. Cada objeto de busca subsequente é comparado com o elemento no topo do $heap$. Se a distância entre o objeto candidato ao objeto de consulta q é menor do que a distância ao objeto no topo do $heap$, o topo será substituído por um novo objeto.

Algoritmo 5.2: kNN no CUDA-LSH

```
1  algorithm [out_ids, out_distances] = kNN(LSH, q, k)
2      float distances[k] := [∞, ..., ∞]
3      int result_set[k] := [0, ..., 0]
4      for i := 1 to L do
5          hashvalue := LSH_i.hash(q)
6          pos := binary_search(LSH_i.keys, hashvalue)
7          for j := pos − λ to pos + λ do
8              id = LSH_i.ids[j]
9              dist = distance(S[id][], q)
10             update_heap(id, dist, distances, result_set, k)
11         end for
12     end for
13     out_ids := result_set
14     out_distances := distances
15 end.
16 void update_heap(id, dist, array distances, array result_set, N)
17     for pos := 1 to N do
18         if distances[pos] > dist then
19             break
20         end if
21     end for
22     if pos < N ∧ result_set[pos] ≠ id then
23         for i := N − 1 to pos do
24             distances[i + 1] := distances[i]
25             result_set[i + 1] := result_set[i]
26         end for
27         distances[pos] := dist
28         result_set[pos] := id
29     end if
30 end.
```

5.2.4 Consultas AllkNN aproximada no CUDA-LSH

Resolver o problema de busca AllkNN não é trivial, especialmente quando se trabalha com grandes conjuntos de dados em altas dimensões. Além disso, os problemas relacionados à execução de consultas kNN eficientes também precisam ser considerados, em especial o problema da "maldição da alta dimensionalidade". Desse modo, baseado em uma abordagem aproximada, o método CUDA-LSH desenvolvido para fornecer buscas kNN eficientes em CUDA (Algoritmo 5.2) foi estendido para consultas AllkNN.

Em razão da natureza altamente paralelizável da consulta AllkNN, é possível superar alguns dos problemas relacionados ao se trabalhar com grandes conjuntos de dados. Assim, apoiado pelo método de indexação CUDA-LSH, foi desenvolvida uma solução massivamente paralela do algoritmo de busca

AllkNN. O pseudocódigo para o problema AllkNN implementado em CUDA é descrito no Algoritmo 5.3. Considerando o fato de que é necessário preparar a estrutura do índice antes de se iniciar o processo de consulta, as linhas de pseudocódigo (2-7) são as mesmas que o algoritmo de indexação descrito no Algoritmo 5.1.

Assim, o passo adicional se dá pela chamada à função $kernel$ AllkNN-kernel (linha 8). Observa-se que a diferença fundamental entre o Algoritmo kNN e o Algoritmo AllkNN se encontra na linha 11, em que um novo id é gerado automaticamente, a fim de processar o mesmo código do algoritmo kNN. Finalmente, o algoritmo AllkNN retorna o id e a distância do elemento s_t ao seu k-ésimo vizinho mais próximo.

Algoritmo 5.3: CUDA-AllkNN no CUDA-LSH

```
1  algorithm CUDA−AllkNN(S, k)
2      deviceMem LSH := initHashFunctions() // inicialização das funções hash
3      deviceMem S'[][] := S[][]
4      threads := nTheads
5      blocks := N/nTheads
6      LSH−Kernel<<<blocks, threads>>>(S', LSH)
7      cuda_sort(LSH.keys, LSH.ids)
8      AllkNN−Kernel<<<blocks, threads>>>(S', LSH, k)
9  end.
10 kernel [out_ids, out_distances] AllkNN−Kernel(S, LSH, k)
11     t := blockIdx.x ∗ blockDim.x + threadIdx.x // point id
12     float query[] := S[t][]
13     float distances[k] := [∞, ..., ∞]
14     int result_set[k] := [0, ..., 0]
15     for i := 1 to L do
16         hashvalue := LSH_i.hash(query)
17         pos := binary_search(LSH_i.keys, hashvalue)
18         for j := pos − λ to pos + λ do
19             id = LSH_i.ids[j]
20             dist = distance(S[id][], query);
21             update_heap(id, dist, distances, result_set, k)
22         end for
23     end for
24     out_distances[t] = distances[k] // distância do elemento s_t ao seu k−ésimo vizinho mais próximo
25     out_ids[t] = t
26 end.
```

Note que o Algoritmo 5.3, para a busca AllkNN no CUDA-LSH, utiliza o mesmo esquema de paralelização do Algoritmo de construção. Desse modo, os identificadores de *threads* são associados, por sua vez, aos identificadores dos objetos (linha 11), ou seja, cada objeto é processado paralelamente por uma *thread*, conseguindo com isso uma solução altamente paralela para o problema de todos os k-vizinhos mais próximos.

5.3 Descoberta de *motifs* em CUDA

Uma vez desenvolvida a solução aproximada de alto desempenho, o método CUDA-LSH foi estendido para dar suporte a algoritmos paralelos de descoberta de *motifs*. Nesse sentido, tanto métodos exatos como aproximados de descoberta de *motifs* são analisados com o objetivo de projetar soluções em CUDA. Com esse foco, na Seção 5.3.1 descreve-se o Algoritmo *CUDA TopK-Motifs* e na Seção 5.3.2 descreve-se o Algoritmo *CUDA RandomProjection*.

5.3.1 Descoberta de *motifs* baseado no algoritmo CUDA-AllkNN

Uma vez desenvolvida a solução de alto desempenho para buscas por similaridade aproximada, os algoritmos para a identificação de *motifs*, especificamente os K padrões mais frequentes do conjunto de dados, podem ser processados eficientemente, inclusive para grandes conjuntos de dados. Diante disso, a partir do esquema de indexação CUDA-LSH, o algoritmo de força bruta para descoberta de *motifs*, apresentado na Seção 3.6.1, foi melhorado usando implementações eficientes e paralelas de buscas AllkNN.

Para entender melhor a definição do problema da procura dos *TopK-Motifs* fundamentados nos k-vizinhos mais próximos, a Figura 5.5 ilustra um exemplo de execução. Nesse exemplo, utiliza-se a definição do *k-Motif* (com $k = 1$) para se definirem os *clusters* candidatos. A relevância do *motif* é definida pela densidade dos *clusters*, ou seja, pelo raio do *cluster* que contém os k elementos mais próximos entre si. No exemplo, quanto mais denso é o *cluster* a intensidade do cor preto é maior.

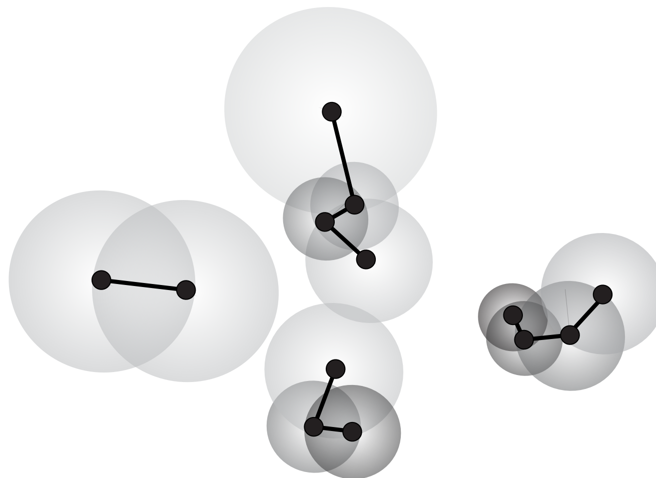


Figura 5.5: Os *clusters* candidatos a *motif* utilizando os vizinhos mais próximos.

O Algoritmo 5.4 descreve o pseudocódigo do algoritmo *CUDA-TopKMotifs* para a identificação dos *TopK-Motifs* que utiliza o algoritmo AllkNN como procedimento principal. Basicamente, a busca kNN do algoritmo de força bruta (Algoritmo 3.1, Seção 3.6.1) é substituída pelo algoritmo de busca paralela fornecido pelo método CUDA-LSH. Note que o algoritmo para a identificação de *motifs*, em um primeiro momento, prepara o índice (linha 6-7), em seguida, o mesmo executa a consulta AllkNN (linha 8), após isso ordena os candidatos baseando-se na densidade dos *clusters* e, finalmente imprime os $topK$ primeiros *motifs* (linha 10-13).

Desse modo, o custo do algoritmo de força bruta é aperfeiçoado em dois sentidos. Primeiro, a busca kNN sequencial é substituída por um algoritmo aproximado de busca por similaridade de custo sublinear fornecido pela técnica CUDA-LSH. Segundo, o processo sequencial que testa todos os possíveis centros de consulta (o primer *loop* no Algoritmo 3.1) é então substituído pela implementação paralela em CUDA do algoritmo AllkNN.

Algoritmo 5.4: Algoritmo CUDA TopK-Motifs

```
1  algorithm CUDA−TopKMotifs(S, k, topK)
2      deviceMem LSH := initHashFunctions() // inicialização das funções hash
3      deviceMem S′[][] := S[][]
4      threads := nTheads
5      blocks := N/nTheads
6      LSH−Kernel<<<blocks, threads>>>(S′, LSH)
7      cuda_sort(LSH.keys, LSH.ids)
8      [ids, distances] := AllkNN−Kernel<<<blocks, threads>>>(S′, LSH, k)
9      cuda_sort(distances, ids)
10     for i := 1 to topK do
11         result := kNN(ids[i], k)
12         print_motif(result)
13     end for
14 end.
```

As etapas do processo de identificação de *motifs* baseado em consultas AllkNN é ilustrado na Figura 5.6. Basicamente, essa figura ilustra as etapas do *pipeline* para o Algoritmo 5.4.

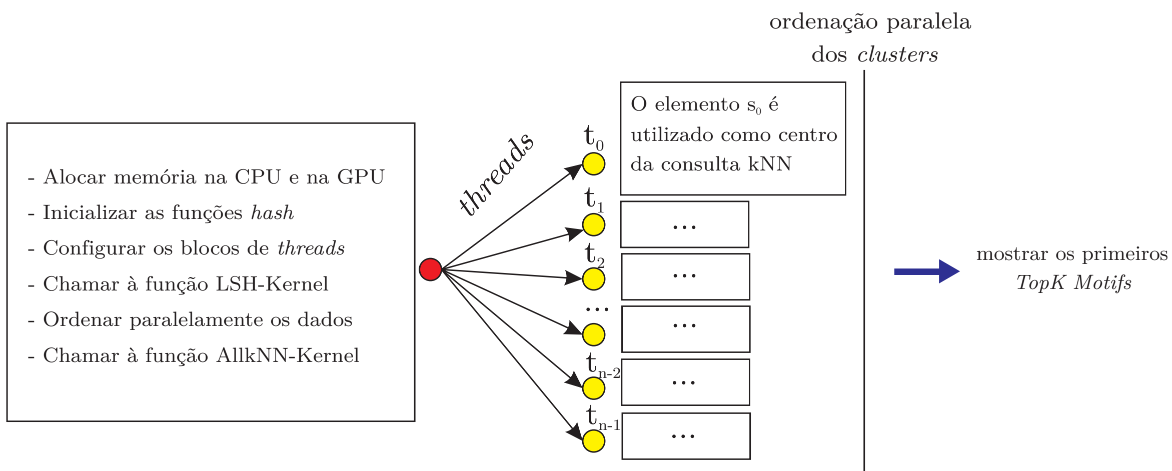


Figura 5.6: Pipeline de consultas kNN paralelas para a identificação dos *motifs*.

5.3.2 *Random Projection* em CUDA

Como explicado na Seção 3.6.3, o algoritmo de descoberta de *motifs* baseado em *Random Projection* tem um custo computacional quadrático, mas tem a vantagem de só processar as palavras SAX. Nesta seção, apresenta-se uma implementação paralela em CUDA desse algoritmo (Algoritmo 3.4, Seção 3.6.3). Uma das vantagens mais importantes do modelo de programação em CUDA, ou seja, a

geração automática dos identificadores para os *threads*, é explorada para conseguir uma solução de alto desempenho para a identificação dos *TopK-Motifs*.

Algoritmo 5.5: Algoritmo CUDA *Random Projection* (versão 1)

```
1  algorithm CUDA−RandomProjection(S)
2      deviceMem W := BuildSAXWords(S)
3      deviceMem MC := Initialize(N, 0)
4      threads := < twidth, twidth > //twidth = 32
5      blocks := < N/twidth, N/twidth >
6      for i = 1 to iterations do
7          deviceMem mask := random(mask_size)
8          RP−Kernel<<<blocks, threads>>>(MC, W, mask)
9      end for
10 end.
11 kernel RP−Kernel(MC, W, mask)
12     i := blockIdx.x ∗ blockDim.x + threadIdx.x
13     j := blockIdx.y ∗ blockDim.y + threadIdx.y
14     if i ≠ j ∧ W_i[mask] = W_j[mask] then
15         MC[i][j]++
16     end if
17 end.
```

O Algoritmo 5.5 apresenta o pseudocódigo associado à implementação do algoritmo *Random Projection* em CUDA. Note que os dois *loops* do Algoritmo *Random Projection* em CPU (linhas 11-12 no Algoritmo 3.4) são agora substituídos por blocos de *threads* (linhas 12-13). O número de blocos é de $N/twidth * N/twidth$ e o número de *threads* por bloco é de $twidth * twidth$, obtendo, desta forma, uma $grid$ de $N * N$ *threads*. Desse modo, cada *thread* da $grid$ de *threads* corresponde a uma iteração no algoritmo em CPU.

No entanto, esse algoritmo pode ser ainda melhorado com o uso de memória compartilhada. Assim, utilizando blocos de *threads* o problema pode ser dividido em várias pequenas partições. O uso de memória compartilhada entre permite combinar as soluções locais, reduzindo assim o custo de comunicação entre blocos. Essa solução é mais elegante, dado que o problema é dividido em subproblemas locais que são combinados depois. O algoritmo melhorado de *Random Projection* em CUDA é descrito no Algoritmo 5.6.

À título de exemplificação, quando $N = 32000$ (número de séries a serem processados) e $twidth = 32$ (bloco de trabalho), as duas versões do algoritmo *CUDA-RandomProjection* gera um número diferente de *threads*. Isto é detalhado a seguir.

***Random Projection* (versão 1)** :

$$\begin{aligned} &< blocks > \times < threads > \\ &=< N/twidth, N/twidth > \times < twidth, twidth > \\ &=< 1000, 1000 > \times < 32, 32 > \\ &= 1024000000 \end{aligned}$$

***Random Projection* (versão 2)** :

$$\begin{aligned} &< blocks > \times < threads > \\ &=< N/twidth, N/twidth > \times < twidth > \\ &=< 1000, 1000 > \times < 32 > \\ &= 32000000 \end{aligned}$$

A primeira versão precisa gerar $N \times N$ *threads*, e no segundo caso o algoritmo melhorado apenas precisa gerar $(N \times N)/twidth$ *threads* o que supõe $twidth$ vezes menos chamadas à função $kernel$. Desse modo, utilizando a memoria compartilhada, pode-se economizar banda da memória global, pois

o número total de *threads* e o acesso a variáveis internas é reduzido por meio das vantagens da memória compartilhada.

Algoritmo 5.6: Algoritmo CUDA *Random Projection* (versão 2)

```
1  algorithm CUDA−RandomProjection(S)
2      deviceMem W := BuildSAXWords(S)
3      deviceMem MC := Initialize(N, 0)
4      blocks := <N/twidth, N/twidth> //twidth = 32
5      threads := twidth
6      for i = 1 to iterations do
7          deviceMem mask := random(mask_size)
8          RP−Kernel<<<blocks, threads>>>(MC, W, mask)
9      end for
10 end.
11 kernel RP−Kernel(MC, W, mask)
12     i := blockIdx.x ∗ blockDim.x + threadIdx.x
13     j := blockIdx.y ∗ blockDim.y
14     __shared__ CudaWord A[nThreads] := W[i][]
15     __shared__ CudaWord B[nThreads] := W[j + threadIdx.x][]
16     __syncthreads()
17    for k := 0 to blockDim.x − 1
18       if A[k] = B[threadIdx.x] then
19          MC[i][j]++
20       end if
21       j++
22    end for
23 end.
```

5.4 Considerações Finais

Neste capítulo foram apresentadas soluções de alto desempenho para o problema de busca por similaridade aproximada e identificação de *motifs*. A principal contribuição deste capítulo foi o desenvolvimento da técnica CUDA-LSH como suporte de algoritmos paralelos de busca AllkNN e descoberta dos *TopK-Motifs*. A técnica CUDA-LSH é uma estrutura de indexação que permite explorar as características multi-núcleos das GPUs para tarefas que são consideradas impraticáveis em muitos contextos, pois são processos que precisam de grandes recursos de computação. Por meio da flexibilização e simplificação dos algoritmos e das estruturas de dados do esquema de indexação LSH foi possível conseguir implementações massivamente paralelas em GPU. A redução da maioria das etapas de computação na GPU foi o critério de projeto mais marcante da técnica CUDA-LSH. Portanto, foi possível diminuir drasticamente a sobrecarga de cópia de variáveis entre GPU e CPU e, consequentemente, melhorar o seu desempenho para problemas complexos de busca e mineração.

Para a avaliação dos algoritmos em CUDA, foi apresentada a técnica de busca AllkNN, implementando-a no algoritmo *CUDA-AllkNN()*, além dos métodos de descoberta de *motifs* implementados nos Algoritmos *CUDA-RandomProjection()* e o *CUDA-TopKMotifs()*.

CAPÍTULO

6

Experimentos

6.1 Considerações Iniciais

Na literatura das áreas de bases de dados e mineração de dados, a abordagem mais comum encontrada para validação de novas técnicas é baseada em estudos experimentais. Como este trabalho tem como principal objetivo investigar e desenvolver soluções de alto desempenho de busca por similaridade aproximada e computação GPGPU para reduzir o custo computacional de tarefas de mineração de dados que utilizam o kNN como procedimento mais importante, em geral os experimentos envolvem:

1. Análises de resultados obtidos com conjuntos de dados reais e sintéticos para mostrar a vantagem dos métodos de busca por similaridade aproximada baseados no LSH frente a métodos exatos mais clássicos, como os Métodos de Acesso Métricos (MAMs). Nesse sentido, compara-se tanto métodos exatos como aproximados que representem o estado da arte, com o objetivo de demonstrar a eficácia e o desempenho das técnicas LSH;

2. Análises de resultados obtidos na etapa anterior para identificar o método de busca aproximada mais apropriado, considerando soluções aproximadas que possam se beneficiar dos recursos de computação para alto desempenho fornecidos pela GPU, para fornecer consultas por similaridade de alto desempenho. Como foi discutido na Seção 1.2, são considerados métodos que garantam um custo sublinear nas consultas, controle da qualidade nos resultados e em especial que possam ser implementados em CUDA;

3. Uma vez identificada e desenvolvida a solução aproximada de alto desempenho, o método CUDA-LSH foi analisado como suporte de algoritmos paralelos de descoberta de *motifs* em séries temporais. Nesse sentido, são comparados métodos de descoberta de *motifs* tanto exatos como aproximados que representem o estado da arte, com o objetivo de demonstrar a eficácia e o desempenho da técnica proposta baseada na implementação da busca AllkNN do CUDA-LSH.

Assim, os experimentos discutidos neste trabalho têm por objetivo demonstrar que o método CUDA-LSH, como suporte de algoritmos paralelos de busca AllkNN e descoberta de *motifs*, apresenta as seguintes características:

1. O algoritmo *CUDA-AllkNN()*, permite identificar todos os vizinhos mais próximos para todos os objetos de consulta;

2. A técnica de descoberta de *motifs*, implementada pelo algoritmo *CUDA-TopKMotifs()* e baseada no *CUDA-AllkNN*, permite identificar os *motifs* relevantes nos dados;

3. A precisão das técnicas propostas é comparável às técnicas exatas com a vantagem de ter um custo sublinear nas consultas tanto para a identificação dos k-vizinhos mais próximos como para a identificação de *motifs*.

É importante mencionar que as soluções propostas neste trabalho foram projetadas para tratar conjuntos de séries temporais que podem ser definidos também como dados multidimensionais compostos por atributos numéricos.

Os experimentos apresentados neste capítulo são divididos em três grupos:

1. Experimentos com dados reais e sintéticos, discutindo o desempenho das consultas, acurácia (*recall*), uso de espaço de memória e escalabilidade dos métodos exatos e aproximados para busca por similaridade aproximada em consultas por abrangência (Seção 6.3.1);

2. Experimentos com dados reais e sintéticos, discutindo o desempenho e acurácia dos métodos de indexação em consultas kNN e AllkNN, incluindo as soluções paralelas em CPU e GPU do algoritmo *CUDA-AllkNN()* (Seção 6.3.2);

3. Experimentos com dados reais e sintéticos, discutindo o desempenho e a acurácia dos métodos exatos e aproximados para descoberta de *motifs*, incluindo o algoritmo *CUDA-TopKMotifs()* (Seção 6.3.3).

6.2 Materiais e Métodos

Para detalhar o processo de análise tanto dos vizinhos mais próximos como dos *motifs* encontrados utilizando as técnicas propostas, são avaliados os resultados de estudos experimentais realizados com 10 conjuntos de dados sintéticos e reais. A lista a seguir descreve os conjuntos de dados utilizados.

1. **SYNT16** Este conjunto de dados sintéticos contém 10.000 vetores de dimensão 16, distribuídos uniformemente em grupos de 10 em um hipercubo 16-d.

2. **SYNT32** Semelhante ao SYNT16, mas contém 100.0000 vetores de dimensão 32.

3. **SYNT64** Semelhante ao SYNT16, mas contém 10.000 vetores de dimensão 64.

4. **SYNT256** Semelhante ao SYNT16, mas contém 10.000 vetores de dimensão 256.

5. **COLOR** Este conjunto de dados reais contém 68.000 vetores de dimensão 32. Cada vetor descreve o histograma de cores de uma imagem em um conjunto de dados da coleção Corel [1].

[1] http://kdd.ics.uci.edu/databases/CorelFeatures/

6. **MNIST** Este conjunto de dados reais contém 60.000 vetores de dimensão 50. O conjunto de dados MNIST[2] é composto por dígitos manuscritos e é um subconjunto de um conjunto maior de dados disponíveis do NIST (*National Institute of Standards and Technology*). A dimensionalidade é reduzida, possuindo as 50 dimensões com as maiores variações.

7. **AUDIO** Este conjunto de dados reais contém 54.387 vetores de dimensão 192. O conjunto de dados de áudio vem da coleção LDC SWITCHBOARD-1[3]. Esta coleção de séries temporais tem cerca de 2400 conversas telefônicas entre dois lados de 543 palestrantes de todas as áreas dos Estados Unidos.

8. **EOG** O conjunto de dados EOG contém 100.000 vetores de dimensão 256. O conjunto de dados EOG[4] é composto por valores temporais de movimentos oculares com uma taxa de amostragem de 250.

9. **RWALK** Este conjunto de dado é composto por séries aleatórias de 100.000 elementos de comprimento 32. As séries temporais foram reproduzidas, seguindo as instruções no site MK-motif [5], utilizando a mesma semente aleatória.

10. **AGRODATA** Este conjunto de dados é fornecido por pesquisadores colaboradores do Centro de Pesquisas Metereológicas e Climáticas Aplicadas à Agricultura (Cepagri - Unicamp), e da Embrapa Informática Agropecuária de Campinas, no contexto do projeto AgroDataMine[6]. Estas séries temporais climáticas foram fornecidas originalmente pelo Agritempo[7]. Este conjunto de dados contém 100.000 vetores de dimensão 24 correspondentes às medidas diárias de temperatura média coletadas por 24 estações meteorológicas localizadas no estado de São Paulo, Brasil, no período 1961-1990.

Os principais aspectos relacionados aos conjuntos de dados foram:

- Os conjuntos de dados foram selecionados por serem utilizados na literatura para testar métodos de indexação e descoberta de *motifs*. Por exemplo, foi adotado um processo padrão para gerar os conjuntos de dados sintéticos (1 a 4), como descrito em [Ciaccia et al., 1997]. Estes conjuntos foram utilizados devido à sua simplicidade para criar cenários complexos. Os conjuntos de dados reais COLOR, MNIST e AUDIO já foram utilizados para testar os métodos LSH em [Datar et al., 2004, Lv et al., 2007, Tao et al., 2010];

- Dado que muitos dos métodos para a identificação de *motifs*, especificamente os baseadas em representações SAX, exploram a natureza temporal das séries temporais, apenas os conjuntos de dados AUDIO, EOG, RWALK e AGRODATA foram considerados para avaliar o desempenho dos algoritmos para descoberta de *motifs*;

[2] http://yann.lecun.com/exdb/MNIST/
[3] http://www.ldc.upenn.edu/Catalog/docs/switchboard/
[4] http://www.cs.ucr.edu/~mueen/OnlineMotif/index.html
[5] http://www.cs.ucr.edu/~mueen/MK/
[6] http://gbdi.icmc.usp.br/agrodatamine/
[7] http://www.agritempo.gov.br/

- O conjunto de dados AGRODATA inclui dados reais cuja análise é de interesse dos pesquisadores envolvidos no projeto “AgroDataMine: Desenvolvimento de Métodos e Técnicas de Mineração de Dados para Apoiar Pesquisas em Mudanças Climáticas com Ênfase em Agrometeorologia” [8], em andamento no GBDI (Grupo de Bases de Dados e Imagens);

- Nos experimentos deste trabalho, os conjuntos de teste com os objetos de consulta são criados para cada um dos conjuntos de dados usado 500 objetos escolhidos aleatoriamente do conjunto de dados original. Metade deles (250) foram retirados do conjunto de dados antes de criar os índices. Esta configuração permitirá avaliar os algoritmos com centros de consulta tanto dentro do índice como fora deste.

A configuração do computador usado para os experimentos foi: uma CPU Intel core i7 2.67GHz com 6 Gb de memória RAM, com uma GPU Geforce GTX 470 com 1 Gb de VRAM. O Sistema operacional foi o Microsoft Windows 7, e os compiladores usados foram o Microsoft Visual Studio 2008 (cl) e o compilador CUDA da NVIDIA (nvcc). A fim de obter uma comparação justa, todos os métodos de indexação e de descoberta de *motifs* foram implementados em C++, todos com o mesmo código de otimização. Alguns experimentos, especificamente para o tipo de consulta a todos os k-vizinhos mais próximos (AllkNN), incluíram involucram a comparação entre implementações *multi-threads* tanto para CPU como para GPU. Realiza-se assim uma avaliação tanto quantitativa como qualitativa na modelagem do comportamento de algoritmos eficientes tanto em CPU como em GPU. A Tabela 6.1 apresenta as especificações do computador e dos compiladores utilizados.

Tabela 6.1: Especificações da CPU e GPU assim como os compiladores utilizados nos experimentos.

Processador	**Tipo**	**# de Núcleos**	**# de *Threads***	**RAM**	**Compilador**
Intel Core i7 2.67GHz	CPU	4	2	6 GB	cl
Intel Core i7 2.67GHz	CPU	4	4	6 GB	cl
Intel Core i7 2.67GHz	CPU	4	8	6 GB	cl
NVIDIA Geforce GTX 470	GPU	448	1024	1 GB	nvcc 4.0

Além disso, varias métricas para a avaliação de desempenho e acurácia (ou precisão) dos métodos foram usados, como discutido a seguir.

Os métodos de busca por similaridade e de descoberta de *motifs* implementados foram avaliados usando as seguintes métricas de comparação:

- Consulta por Abrangência exata e aproximada

  - Desempenho nas consultas: neste experimento foi avaliado o desempenho da abordagem proposta em relação a outros índices bem conhecidos em consultas por abrangência. O

[8] Projeto apoiado pelo Instituto Fapesp-Microsoft Research

objetivo deste experimento é medir o número médio de cálculos de distância e o tempo total gasto para recuperar os objetos mais próximos aos objetos de consulta de um conjunto de testes usando consultas Rq. Os métodos foram testados com diferentes valores de r. Assim, os valores dos raios para consultas por abrangência foram definidos em função do valor de objetos desejados por cada consulta (de 1 até 10);

  - Acurácia: neste experimento foi avaliada o *recall* da abordagem proposta em relação a outros métodos de busca aproximada. Dado um conjunto de dados, pode-se avaliar a acurácia do método usando busca sequencial. Assim, para cada consulta verifica-se se a resposta inclui os mesmos elementos retornados por uma busca de varredura sequencial;
  - Uso de espaço: como as técnicas aproximadas precisam de muitos subíndices para garantir resultados de boa qualidade, é necessário medir o custo de memória. Assim, neste experimento foi avaliado o uso de espaço em *megabytes* dos métodos exatos e aproximados, ressaltando que os métodos aproximados garantem um custo sublinear com qualidade nos resultados;
  - Escalabilidade: neste experimento, foi estudado o comportamento das técnicas quando o tamanho do conjunto de dados aumenta. Para medir o desempenho e a escalabilidade das técnicas varia-se o tamanho do conjunto de dados. Desse modo, para cada conjunto foram executadas 500 consultas com diferentes objetos de busca. Como o comportamento é equivalente para diferentes valores de r, são apresentados apenas os resultados para o raio que recupera em média 10 objetos.

- Consultas kNN e AllkNN
  - Desempenho nas consultas: neste experimento foi avaliado o desempenho da abordagem proposta em relação a implementações paralelas de consultas AllkNN. O objetivo deste experimento é medir o tempo total gasto para recuperar os objetos mais próximos aos objetos de consulta de um conjunto de testes usando consultas kNN. O número de vizinhos k a serem encontrados nas consultas kNN foram escolhidos de acordo com os valores frequentes usados em situações reais. Em todos os experimentos foram usados valores diferentes para k, variando de 1 a 100. Devido às limitações de espaço na GPU, no caso das consultas AllkNN o número de vizinhos a serem encontrados foi limitado em 25.
  - Acurácia: idealmente, um sistema de busca por similaridade deve ser capaz de atingir altos níveis de desempenho e qualidade na busca utilizando uma pequena quantidade de espaço. Neste experimento foi avaliada o *recall* da abordagem proposta em relação a outros métodos de busca aproximada. Dado um objeto de consulta q, seja $I(q)$ o conjunto de resposta ideal, isto é, os k-vizinhos mais próximos ao q, e $A(q)$ o conjunto de resposta atual. Para calcular a *recall* se faz uso de:

$$recall = \frac{|A(q) \cap I(q))|}{|I(q)|} \tag{6.1}$$

    No caso ideal, a pontuação é de $1,0$ o que significa que todos os k vizinhos mais próximos são retornados. Para fins de comparação, também são calculadas as taxas de erro

($error_ratio$), ou o erro efetivo, que mede a qualidade das buscas conforme definido em [Gionis et al., 1999].

$$error_ratio = \frac{1}{|Q|K} \sum_{k=1}^{K} \frac{d_{LSH_k}}{d_k^*} \tag{6.2}$$

Onde d_{LSH_k} é a distância do objeto da consulta q ao k-vizinho mais próximo encontrado pelo método LSH, e d_k^* é a distância do objeto da consulta q ao verdadeiro k-vizinho mais próximo. Em outras palavras, mede-se o quão próximas as distâncias dos k vizinhos mais próximos encontrados pelo LSH são das distâncias aos verdadeiros k-vizinhos mais próximos de q.

– Escalabilidade: para avaliar a escalabilidade do método em consultas AllkNN, o CUDA-LSH foi implementado em duas versões: GPU e CPU. No primeiro experimento de escalabilidade mantém-se o tamanho do conjunto de dados enquanto o nível de paralelismo aumenta. No segundo experimento, compara-se os tempos da implementação em CPU e GPU das consultas AllkNN quando o tamanho do conjunto de dados cresce. O desempenho é avaliado mediante o *speedup* entre o tempo de execução utilizando CPU e GPU.

$$speedup = \frac{tempo_CPU}{tempo_GPU} \tag{6.3}$$

- Descoberta de *motifs*

  Para avaliar os métodos de descoberta de *motifs* é apresentada uma avaliação tanto quantitativa como qualitativa para as soluções exatas e aproximadas. O comportamento do método proposto foi comparado com algoritmos probabilísticos e aproximados de identificação de *motifs* encontrados na literatura. Além disso, a plataforma de programação CUDA foi utilizada para criar implementações massivamente paralelas das técnicas analisadas, entre eles o *CUDA-RandomProjection()* (ver Seção 5.3.2) e o *CUDA-TopKMotifs()* (ver Seção 5.3.1). Para a identificação dos *motifs* relevantes, diversos conjuntos de séries temporais reais e sintéticos foram usados. Desse modo, com a finalidade de realizar uma ampla investigação que permitisse analisar os principais aspectos relacionados à metodologia e ao processo de identificação de *motifs*, essa avaliação foi dividida nos seguintes passos:

  – Desempenho: o desempenho dos métodos é avaliado comparando os tempos de execução dos algoritmos e mediante o *speedup* (Equação 6.3) entre o tempo de execução utilizando CPU e GPU.

  – Escalabilidade: para avaliar a escalabilidade dos algoritmos de descoberta de *motifs*, a técnica proposta foi comparada com métodos exatos e aproximados para a identificação do *Pair-Motif* e dos *TopK-Motifs*. Neste experimento, foram comparados os tempos de execução dos algoritmos em CPU e GPU quando o tamanho do conjunto de dados cresce.

  – Precisão: Para avaliar a precisão dos algoritmos de descoberta de *motifs* utiliza-se os *motifs* encontrados e suas informações, especificamente as distâncias entre os elementos que conformam o *motif* como indicadores para calcular a densidade do *cluster*.

Os resultados dos experimentos serão apresentados e discutidos a seguir.

6.3 Resultados e Discussão

Foi realizada uma avaliação tanto quantitativa como qualitativa no comportamento dos algoritmos de busca por similaridade e os algoritmos de descoberta de *motifs*, considerando tanto soluções exatas como aproximadas. Para responder consultas por similaridade aproximada, métodos exatos e aproximados foram comparados em dois tipos de consulta: consulta por abrangência e consulta aos k-vizinhos mais próximos. Para a identificação de *motifs*, foram comparados os métodos focados na descoberta do *Pair-Motif* e dos *TopK-Motifs*.

6.3.1 Consultas por abrangência exata e aproximada

Diversas técnicas de indexação propostas na literatura foram avaliadas em relação ao desempenho global das consultas (número de cálculos de distância e tempo de resposta), acurácia, uso de espaço e escalabilidade. O desempenho dos métodos LSH, ou seja, LSH clássico [Datar et al., 2004], LSH Multi-probe [Lv et al., 2007] e LSH Multi-level [Ocsa e Sousa, 2010], foi comparado ao desempenho das estruturas métricas Slim-Tree [Traina C. et al., 2002], DF-Tree [Traina et al., 2002] e DBM-Tree [Vieira et al., 2004].

Tabela 6.2: Parâmetros LSH para os conjuntos de dados SYNT16, SYNT64, SYNT256, MNIST e AUDIO.

Parâmetros LSH		
Conjunto de Dados	Método	Parâmetros
SYNT16 (16-D)	LSH	$L = 10, m = 8$
	Multi-probe LSH	$L = 3, m = 8, \tau = 20$
	Multi-level LSH	$L = 3, C = 64$
SYNT32 (32-D)	LSH	$L = 135, m = 24$
	Multi-probe LSH	$L = 14, m = 10, \tau = 30$
	Multi-level LSH	$L = 17, C = 64$
SYNT64 (64-D)	LSH	$L = 54, m = 10$
	Multi-probe LSH	$L = 8, m = 10, \tau = 30$
	Multi-level LSH	$L = 8, C = 64$
SYNT256 (256-D)	LSH	$L = 231, m = 16$
	Multi-probe LSH	$L = 40, m = 16, \tau = 40$
	Multi-level LSH	$L = 40, C = 256$
COLOR (32-D)	LSH	$L = 153, m = 14$
	Multi-probe LSH	$L = 35, m = 14, \tau = 20$
	Multi-level LSH	$L = 35, C = 128$
MNIST (50-D)	LSH	$L = 231, m = 16$
	Multi-probe LSH	$L = 37, m = 16, \tau = 30$
	Multi-level LSH	$L = 37, C = 128$
AUDIO (192-D)	LSH	$L = 62, m = 20$
	Multi-probe LSH	$L = 10, m = 20, \tau = 40$
	Multi-level LSH	$L = 10, C = 256$

Para os métodos LSH observou-se resultados eficientes quando valores adequados para os parâmetros m (número de funções *hash*), L (número de subíndices), τ (número de $probes$) são escolhidos. Os parâmetros LSH (m e L) utilizados neste experimento foram ajustados de acordo com a implementação

E2LSH [9], em função do conjunto de dados, para minimizar o tempo de execução das consultas mantendo sempre a solução dentro dos limites da memória principal. O parâmetro τ é definido usando o seguinte raciocínio: o LSH original usa L projeções, uma para cada subíndice, mas o LSH Multi-probe requer menos subíndices para responder uma consulta (digamos L', $L' < L$); assim, o número de projeções utilizado pelo LSH Multi-probe é $L' \times \tau$, que deve ser aproximadamente igual a L. No entanto, este raciocínio nem sempre é exato (por exemplo, no caso de conjuntos de dados não uniformes) e, portanto, para alguns conjuntos de dados tanto o parâmetro τ como a capacidade nos *buckets* C foram escolhidos empiricamente, a fim de atingir uma acurácia comparável. A Tabela 6.2 apresenta os parâmetros LSH utilizados nestes experimentos. Por simplicidade, em todos os experimentos o tamanho da página de todas as estruturas métricas foi configurado para manter 64 objetos por nó.

Experimento 1: Desempenho nas consultas

A seguir são descritos os resultados de desempenho, medidos em termos de número médio de cálculos de distância e tempo total, para consultas por abrangência nos conjuntos de dados sintéticos SYNT16, SYNT64 e SYNT256 e nos conjuntos de dados reais MNIST, COLOR e AUDIO. Considerando os métodos exatos DBM-Tree, DF-Tree, Slim-Tree e aproximados LSH, LSH Multi-level, LSH Multi-probe, os resultados são resumidos nas Figuras 6.1 e 6.2 e nas Tabelas 6.3 e 6.4.

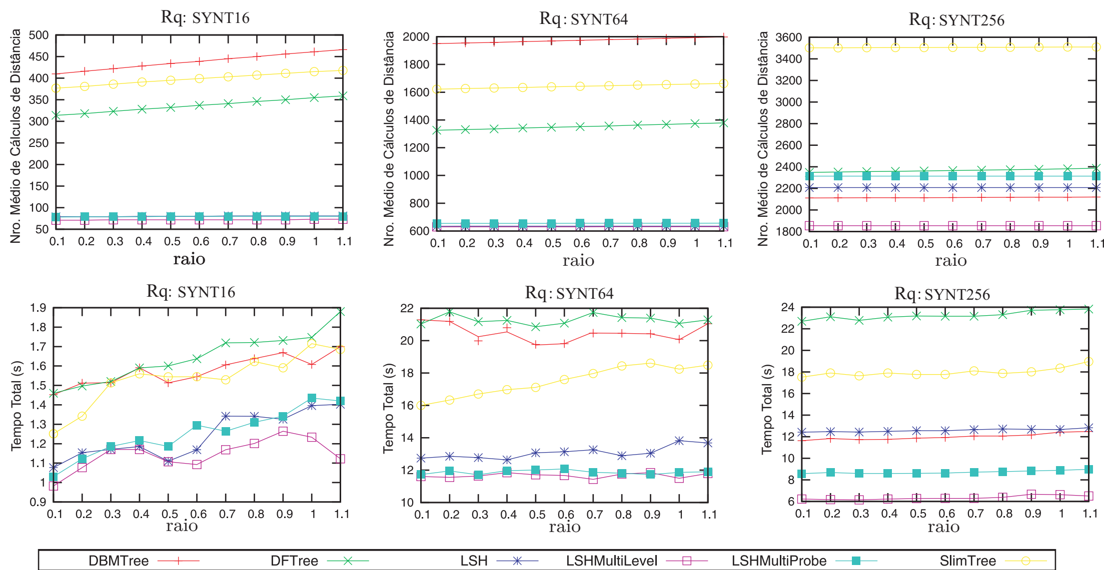


Figura 6.1: Comparação de resultados em consultas por abrangência (Rq) usando a média do número de cálculos de distância (primeira linha) e tempo de resposta (segunda linha) para os conjuntos de dados SYNT16 (primeira coluna), SYNT64 (segunda coluna) e SYNT256 (terceira coluna).

Como pode ser observado nas Figuras 6.1 (dados sintéticos) e 6.2 (dados reais) os métodos LSH são mais rápidos do que as árvores métricas independente da dimensão ou o tamanho do conjunto de dados. Isto é esperado já que as estruturas hierárquicas sofrem da “maldição da alta dimensionalidade”, o que implica sobreposição entre as regiões, quando a dimensionalidade é muito alta e, como consequência, eles precisam explorar muitos caminhos na estrutura das árvores durante o processo de consulta. Por outro lado, os métodos LSH não têm esses problemas, pois só mapeiam os objetos de consulta no espaço

[9] http://www.mit.edu/~andoni/LSH/

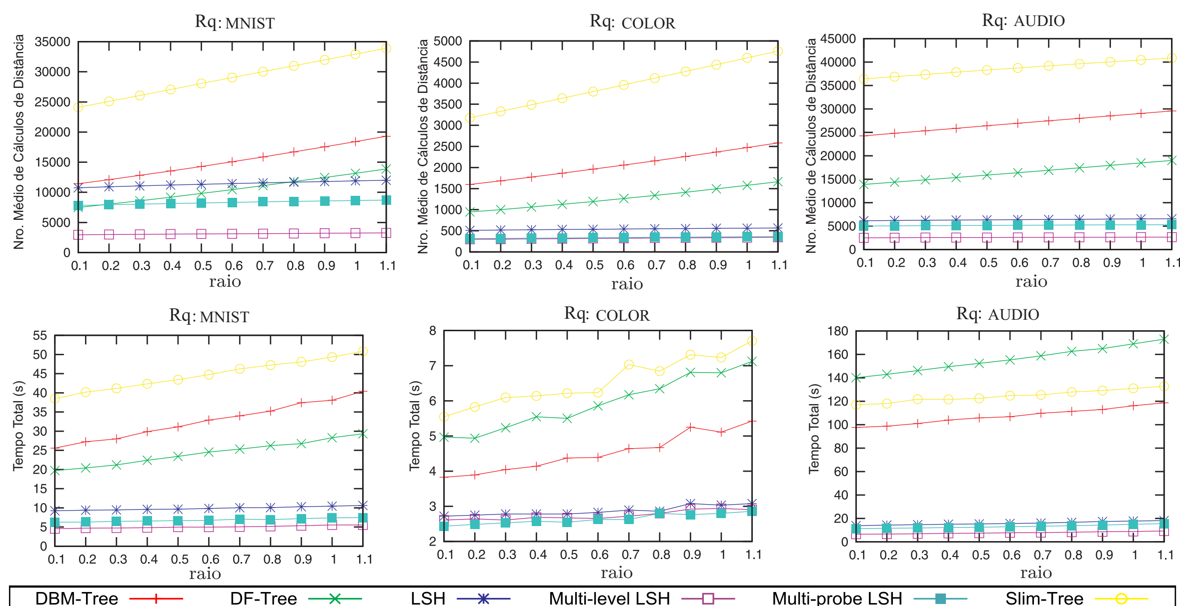


Figura 6.2: Comparação de resultados em consultas por abrangência (Rq) usando a média do número de cálculos de distância (primeira linha) e tempo de resposta (segunda linha) para os conjuntos de dados MNIST (primeira coluna), COLOR (segunda coluna) e AUDIO (terceira coluna).

definidos pelas funções *hash*. No entanto, há um *trade-off* entre o espaço, velocidade e qualidade. Por exemplo, o método LSH clássico garante boa qualidade nos resultados, mas requer ajustes dos parâmetros de domínio para um desempenho ótimo e na maioria dos casos, isso implica o uso de muito espaço em memória.

Tabela 6.3: Comparação do número de cálculos de distância (NCD) para os métodos LSH. Resultados para os conjuntos de dados COLOR (32-D), MNIST (50-D), AUDIO (190-D) e SYNT256 (265-D).

	COLOR		**MNIST**		**AUDIO**		**SYNT256**	
	NCD	%ganho	NCD	%ganho	NCD	%ganho	NCD	%ganho
LSH	541	0,00%	11.427	0,00%	6.373,36	0,00%	2.207	4,58%
LSH Multi-probe	340	37,12%	8.310	27,28%	5.175,18	18,80%	2.313	0,00%
LSH Multi-level	322	40,38%	3.116	72,73%	2.588,36	59,39%	1.854	19,84%

Tabela 6.4: Comparação do tempo de resposta (TT) para os métodos LSH. Resultados para os conjuntos de dados COLOR (32-D), MNIST (50-D), AUDIO (190-D) e SYNT256 (265-D).

	COLOR		**MNIST**		**AUDIO**		**SYNT256**	
	TT	%ganho	TT	%ganho	TT	%ganho	TT	%ganho
LSH	2,87	0,00%	9,88	0,00%	15,892	0,00%	12,588	0,00%
LSH Multi-probe	2,64	8,01%	6,82	30,94%	13,133	17,36%	8,715	30,77%
LSH Multi-level	2,74	4,53%	5,05	48,94%	7,776	51,07%	6,345	49,59%

Ao se comparar os métodos aproximados de busca para os conjuntos de dados reais e para o conjunto sintético de maior dimensionalidade, o resultado mais expressivo é para o método LSH Multi-level, como pode ser observado nas Tabelas 6.3 (número de cálculos de distância) e 6.4 (tempo total). Desse modo quando comparado com o LSH original, a redução do número de cálculos de distância chega a ser 72%. Em relação ao tempo total, a redução chega a ser 51%.

Experimento 2: Acurácia

Para avaliar o acurácia da busca, os procedimentos usados foram semelhantes ao Experimento 1. Dado que os experimentos de acurácia são mais relevantes para conjuntos de dados reais, os experimentos foram realizados apenas para os conjuntos de dados COLOR, MNIST e AUDIO. Considerando os métodos LSH, LSH Multi-probe e LSH Multi-level os resultados de acurácia (*recall*) e do número de cálculos de distância são resumidos na Tabela 6.5.

Devido a que as árvores métricas reportam resultados exatos, apenas os resultados de acurácia são reportados para os métodos LSH. Isto ocorre porque a função de distância utilizada neste experimento define um espaço adequado para realizar consultas por similaridade em espaços métricos. Em contraste, os métodos LSH são baseados em técnicas aproximadas de busca e só reportam resultados de qualidade para consultas por abrangência aproximada com raios de busca no intervalo $(1 + \epsilon) * r$.

Como pode ser observado na Tabela 6.5 os métodos LSH apresentam resultados de acurácia acima de 90%. Estes estudos experimentais mostram que embora os métodos de LSH apresentem algumas dificuldades na definição dos parâmetros de domínio, eles ainda são eficazes é mais rápidos do que as árvores métricas especialmente em cenários onde a dimensão dos conjuntos de dados são elevados.

Tabela 6.5: Comparação de acurácia (*recall*) e do número de cálculos de distância (NCD) para os conjuntos de dados COLOR (32-D), MNIST (50-D) e AUDIO (190-D).

	COLOR			**MNIST**			**AUDIO**		
	NCD	%ganho	*recall*	NCD	%ganho	*recall*	NCD	%ganho	*recall*
LSH	541	0,00%	0,99	11.427	0,00%	1,00	6.373	0,00%	0,99
LSH Multi-probe	340	37,12%	0,99	8.310	27,28%	1,00	5.175	18,80%	0,99
LSH Multi-level	322	40,38%	0,99	3.116	72,73%	0,995	2.588	59,39%	0,94

Por outro lado, mesmo que os métodos LSH conseguem melhorar a qualidade dos resultados com a abordagem dos $probes$, a qualidade ea eficiencia tem um certo limite para raios maiores em consultas por abrangência e para valores de k muito altos em consultas kNN. Esse problema será analisado nos experimentos de escalabilidade.

Experimento 3: Uso do Espaço

Na Tabela 6.6 é apresentado o uso da memória em *megabytes* do LSH, LSH Multi-probe, LSH Multi-leve, Slim-tree e do DF-Tree. Somente a árvores SlimTree e a árvore DF-Tree são consideradas para este experimento, pois elas têm o espaço de utilização mínima e máxima entre as árvores analisadas. Considera-se o número de *buckets* e o número de nós como medidas para calcular o uso de espaço de memória.

O LSH precisa de mais memória do que as árvores métricas, no entanto, o LSH Multi-level e o LSH Multi-probe reduzem o uso do espaço em até 35% em comparação com o LSH original. O que é esperado pois ambas técnicas utilizam um número menor de subíndices, já que exploram a abordagem Multi-probe. Uma observação interessante sobre LSH Multi-probe, Multi-level LSH e DF-Tree é que eles usam uma quantidade muito semelhante de memória.

Tabela 6.6: Comparação do uso da memória pelo LSH, LSH Multi-probe, LSH Multi-level, Slim-Tree, e DF-Tree em *megabytes*.

Técnica / Conjunto de Dados	SYNT16	SYNT64	SYNT256	MNIST	COLOR	AUDIO
LSH	12	58	252	740	1404	1428
LSH Multi-probe	3	14	63	185	351	376
LSH Multi-level	3	15	88	235	352	546
Slim-Tree	2	8	47	29	23	94
DF-Tree	3	23	95	28	38	258

Experimento 4: Escalabilidade

O comportamento dos métodos aproximados em relação ao tamanho do conjunto foi avaliado nesta última bateria de testes para as consultas por abrangência. Os resultados das consultas em relação ao número de cálculos de distância, tempo de total e acurácia (*recall*) para o conjunto de dados SYNT32 são apresentados na Figura 6.3. Neste experimento, o conjunto de dados SYNT32 foi dividido em 10. Após inserir cada subconjunto foram executadas 500 consultas por similaridade para o raio que recupera em média 10 objetos.

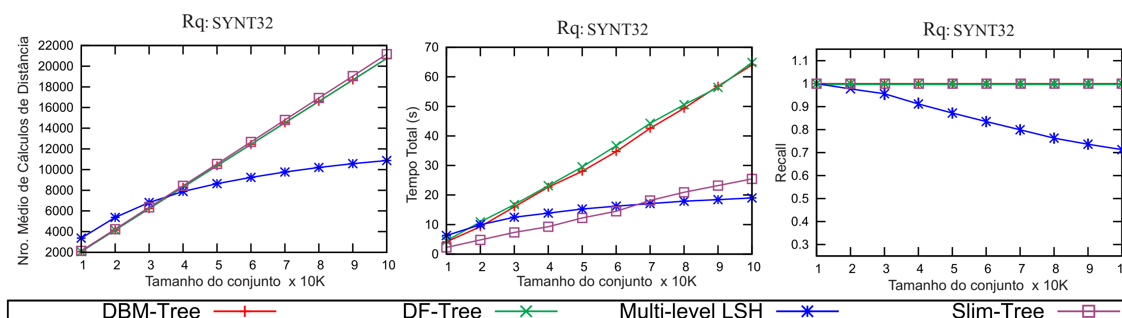


Figura 6.3: Comparação de resultados para consultas por abrangência. Mostra-se o número de cálculos de distância (primeira linha), o tempo de resposta (segunda linha) e o *recall* (terceira linha).

Note que neste experimento o LSH Multi-level foi testado como método representativo dos métodos aproximados, pois até agora apresentou os melhores resultados. Como pode ser observado, os métodos comparados apresentam um comportamento linear em função do tamanho do conjunto de dados, destacando-se em desempenho o método LSH Multi-level, pois tem um custo sublinear nas consultas. No entanto como pode ser observado nos experimentos de acurácia , com o objetivo de manter a qualidade nos resultados em consultas por abrangência pode ser necessário criar mais subíndices conforme o tamanho do conjunto de dados cresce. Este problema pode-se agravar para consultas aos k-vizinhos mais próximos.

Como foi discutido na Seção 2.5.2, mesmo sendo possível adaptar as consultas por abrangência aproximada para responder consultas kNN aproximadas, a qualidade e eficiência tem certo limite para valores grandes de k, pois o LSH garante resultados de qualidade previsível apenas para consultas por abrangência aproximada.

Para atenuar essas dificuldades muitas extensões do método LSH foram propostas, mas para manter alta qualidade nas respostas precisam-se configurar parâmetros de domínio antes de preparar o índice. Para resolver essa dificuldade consideram-se outras abordagens como o método CUDA-LSH, baseado

no Hashfile, como detalhado na próxima seção.

6.3.2 Consultas kNN e AllkNN aproximadas

Após a verificação da eficiência dos métodos baseados em LSH para consultas por abrangência, apresentado na Seção 6.3.1, agora estuda-se o desempenho das técnicas LSH Multi-probe, LSH Multi-level e CUDA-LSH em consultas kNN. Em particular, estamos interessados em avaliar o desempenho do método proposto, o CUDA-LSH, em consultas kNN e AllkNN. O LSH básico e os MAMs foram omitidos porque estes incorrem em um alto custo em espaço (no caso do LSH básico) ou apresentam tempos de consulta não comparáveis ao LSH (no caso dos MAMs).

A avaliação de desempenho foram realizados em diferentes configurações, envolveu conjuntos de dados reais, onde as principais diferenças foram a dimensão, o tamanho dos objetos e do conjunto de dados, a função da distância usada, e a distribuição dos dados. Além disso, considera-se duas maneiras de implementação do CUDA-LSH: em CPU e em GPU.

Os parâmetros de domínio, para os métodos LSH, foram os mesmos utilizados nos experimentos de avaliação de consultas por abrangência. No caso do CUDA-LSH utilizou-se a definição de buscas por similaridade apresentada na Seção 5.2. Assim, os parâmetros de domínio foram configurados com $L = 1$ e $\lambda = 1024$ para as consultas a todos os 25-vizinhos mais próximos. Em geral, para manter resultados de qualidade na maioria de consultas precisa-se só aumentar o valor do parâmetro λ. No entanto, nas consultas aos 100-vizinhos mais próximos, para aumentar o nível de acurácia o número de subíndices foi aumentado a $L = 10$.

Experimento 1: Desempenho dos métodos em consultas kNN

A seguir são descritos os resultados de desempenho, medidos em termos de tempo total, acurácia (*recall*), taxas de erro (*error ratio*) e uso de espaço, para consultas kNN aproximada nos conjuntos de dados COLOR e MNIST. Com o intuito de testar a eficácia dos métodos em diferentes espaços de busca, foram utilizadas as funções de distância L_1 e L_2 para os conjuntos COLOR e MNIST respectivamente.

Desse modo, considerando os métodos CUDA-LSH, LSH Multi-probe e LSH Multi-level, os resultados para consultas aos 100-vizinhos mais próximos são resumidos na Tabela 6.7. Como pode ser observado, quando os parâmetros de domínio são configurados eficazmente os tempos necessários para responder consultas kNN com os métodos LSH são comparáveis. Os resultados de acurácia para L_2 são similares, mas para L_1 apenas o CUDA-LSH e o LSH Multi-level apresentaram bons resultados. Isto é esperado, pois o CUDA-LSH é baseado no HashFile. Como apresentado em [Zhang et al., 2011], o HashFile retorna resultados exatos para a função de distância L_1 e aproximados para a função de distância L_2. Outra observação interessante pode ser feita com relação ao uso de espaço em memória. O uso de espaço em memória do CUDA-LSH é linear, e ao se comparar com os outros métodos, o CUDA-LSH necessita até 10 vezes menos espaço em memória.

Em um outro experimento, como pode ser observado na Figura 6.4, quando o número de vizinhos recuperados varia de 1 a 100, o mesmo comportamento em relação as taxas de erro (*error ratio*) observado na Tabela 6.7 é mantido conforme o número de vizinhos cresce. Note que esta figura ilustra a relação das taxas de erro médio (*error ratio*) para consultas kNN como uma função de k para os conjuntos de dados COLOR e MNIST.

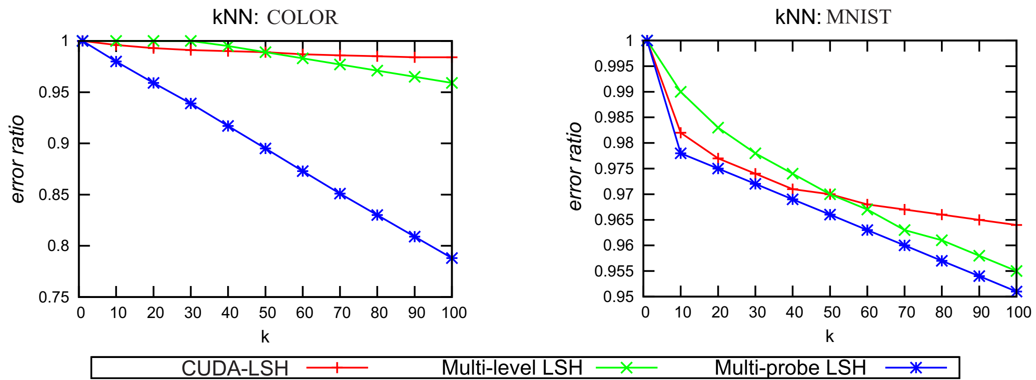


Figura 6.4: Comparação das taxas de error (*error ratio*) para consultas kNN nos conjuntos de dados COLOR (L_1) e MNIST (L_2).

Os resultados mostram que o desempenho do CUDA-LSH é comparável aos métodos LSH, porém requer menos espaço em memória e menos ajuste de parâmetros de domínio, pois na maioria de casos apenas precisa-se ajustar o intervalo de busca λ para melhorar a qualidade dos resultados.

Experimento 2: Desempenho do CUDA-LSH em consultas AllkNN

Para avaliar a desempenho do método em consultas AllkNN, o CUDA-LSH foi implementado em diferentes configurações: em CPU (1, 2, 4 e 8 *threads*) e em GPU (*CUDA-AllkNN*). Como o foco deste trabalho é o desenvolvimento de soluções de alto desempenho para algoritmos de descoberta de *motifs* baseados em soluções aproximadas de busca por similaridade, a escalabilidade da técnica aproximada é um fator importante. Por isso, apenas experimentos de escalabilidade são mostrados nesta seção.

Assim, para o primeiro experimento de escalabilidade mantém-se o tamanho do conjunto de dados, enquanto o nível de paralelismo aumenta. A Figura 6.5 apresenta o tempo necessário para indexar e responder a consulta AllkNN em suas diferentes configurações. A entrada consiste do conjunto de dados SYNTH32 que é composto por 100 mil elementos e o algoritmo foi configurado para recuperar os 25 vizinhos mais próximos.

Como pode ser observado na Figura 6.5 o tempo de execução do algoritmo diminui de 86,43s (configuração de 1 *thread*) para 24,54s (configuração de 8 *threads*) e 13,44s para a versão em GPU. Desse modo ao se comparar com a implementação de 1 *thread* em CPU, a implementação em GPU apresenta uma aceleração de 6,43 vezes (6,43x). Esta aceleração linear é devido ao balanceamento da carga dinâmica das *threads*, como mostrado na Seção 5.2.

Tabela 6.7: Comparação do tempo de consulta, *recall*, *error ratio* e uso de espaço do índice para consultas aproximadas 100-NN. Apresenta-se os resultados para os conjuntos de dados COLOR (L_1) e MNIST (L_2).

	COLOR				MNIST			
	tempo (s)	*recall*	*error ratio*	mem. (Mb)	tempo (s)	*recall*	*error ratio*	mem. (Mb)
CUDA-LSH	**0,005**	**0,92**	**0,98**	**25**	**0,010**	**0,75**	**0,97**	**30**
LSH Multi-probe	0,004	0,60	0,79	351	0,012	0,70	0,95	185
LSH Multi-level	0,004	0,90	0,97	352	0,014	0,72	0,96	235

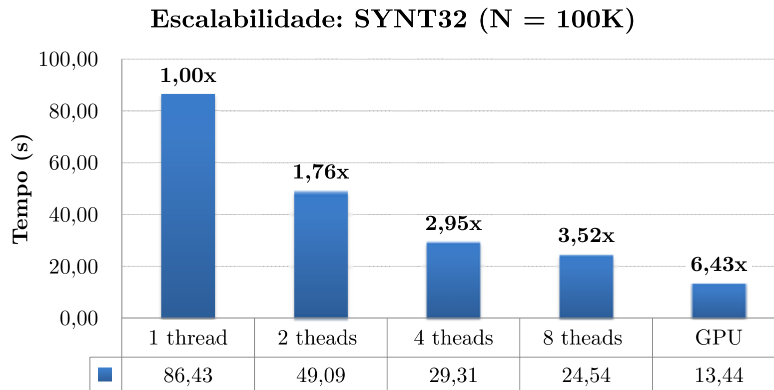


Figura 6.5: Comparação do tempo total de indexação e consulta AllkNN para o método CUDA-LSH em suas diferentes configurações.

Para um segundo experimento de escalabilidade foi testada a implementação do algoritmo AllkNN em CPU com 8 *threads* (Par-AllkNN) e a implementação do algoritmo AllkNN em GPU (CUDA-AllkNN). A comparação dos resultados em termos de tempos de consulta conforme o tamanho no conjunto de dados cresce para os algoritmos CUDA-AllkNN, Par-AllkNN, e AllkNN são resumidos na Figura 6.6. Para esse experimento, divide-se o conjunto de dados SYNT32 em subconjuntos de 10.000 elementos. Após inserir cada subconjunto, executa-se uma consulta AllkNN. Para acelerar as consultas kNN individuais dos algoritmos AllkNN, tanto os implementados em CPU como em GPU são apoiados pelo método CUDA-LSH.

Como pode ser observado na Figura 6.6 a versão CUDA-AllkNN foi de longe a mais rápida entre eles. Esses resultados demonstram o potencial da plataforma de programação CUDA para acelerar consultas por similaridade apoiadas por algoritmos e estruturas de dados simples de ser projetados em GPU. Os resultados também demonstram que embora o tempo total de execução da implementação em CUDA aumenta com o tamanho do conjunto de dados, o aumento no tempo é muito mais linear do que os algoritmos implementados sequencialmente e paralelamente em CPU, que têm grandes saltos entre os tempos de execução.

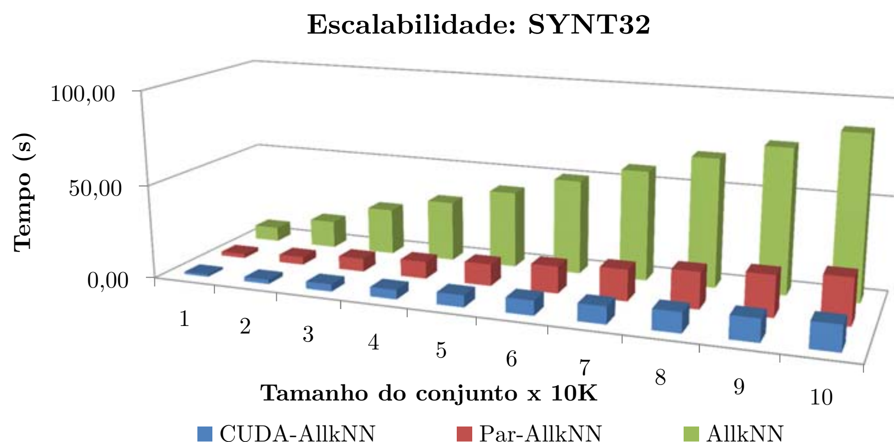


Figura 6.6: Comparação de consultas AllkNN conforme o tamanho do conjunto de dados aumenta. O tempo total de indexação e consulta para o conjunto de dados SYNT32.

6.3.3 Descoberta de *motifs*

Nesta seção são apresentadas as avaliações experimentais realizadas com o objetivo de investigar a eficiência dos métodos de extração de conhecimento em séries temporais, especificamente algoritmos exatos e aproximados para tarefas de descoberta de *motifs*, descrita no Capítulo 3. Desse modo, com a finalidade de investigar os principais aspectos de desempenho relacionados ao processo de identificação de *motifs*, essa avaliação foi dividida nos passos a seguir:

Em primeiro lugar, algoritmos para a identificação do *Pair-Motif* foram avaliados. Desse modo, os algoritmo MK (ver Seção 3.6.2) e *CUDA-TopKMotifs()* (ver Seção 5.3.1) são comparados com o objetivo de avaliar sua eficiência e precisão.

Finalizando, alguns dos mais recentes e conhecidos algoritmos probabilísticos (*Random-Projection*) e aproximados (*MrMotif*) para a identificação de *motifs* são comparados com o algoritmo proposto (*CUDA-TopKMotifs()*). A plataforma de programação CUDA foi utilizada para criar implementações massivamente paralelas das técnicas analisadas, entre eles o *CUDA-RandomProjection* (ver Seção 5.3.2) e o *CUDA-TopKMotifs* (ver Seção 5.3.1).

Experimento 1: Descoberta do *Pair-Motif*

A seguir são descritos os resultados de tempo de processamento para a identificação do *Pair-Motif* nos conjuntos de dados AGRODATA, RWALK, AUDIO e EOG. Considerando os algoritmos MK (ver Seção 3.6.2) e *CUDA-TopKMotifs* (ver Seção 5.3.1), os resultados são resumidos na Figura 6.7. Note que o Algoritmo *CUDA-TopKMotifs()* procura o primeiro *motif* utilizando uma versão restringida da busca AllkNN, em que se procura todos os vizinhos mais próximos a cada elemento do conjunto de dados.

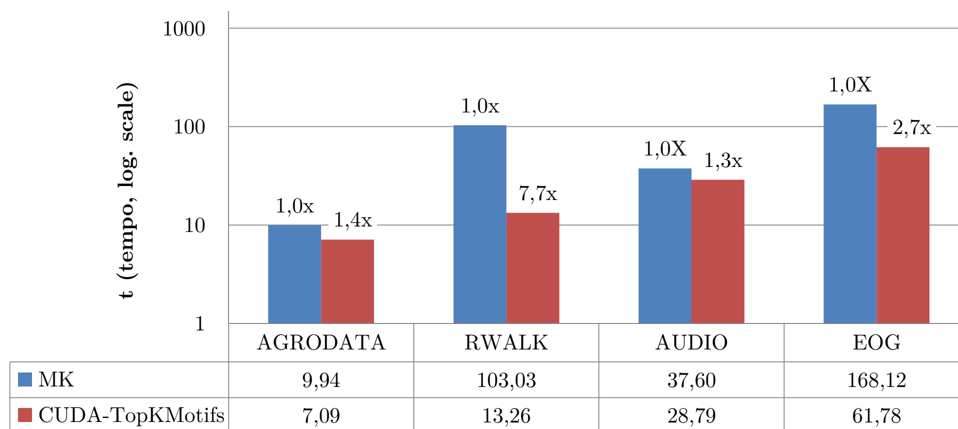


Figura 6.7: Comparação dos tempos de execução dos algoritmos MK e *CUDA-TopKMotif* para a identificação do *Pair-Motif*.

Como pode ser observado na Figura 6.7 os tempos de execução do algoritmo *CUDA-TopKMotifs()*, para os conjuntos de dados AGRODATA, RWALK, AUDIO e EOG, são melhores do que a implementação sequencial do algoritmo MK. Por exemplo, a aceleração, no caso da implementação em GPU é de até 7,7x para o conjunto de dados RWALK.

Verifica-se que devido à precisão para consultas kNN fornecida pelo CUDA-LSH com valores de k menores que 10 (ver Figura 6.4), os resultados na identificação *Pair-Motif* foram exatos na maioria dos

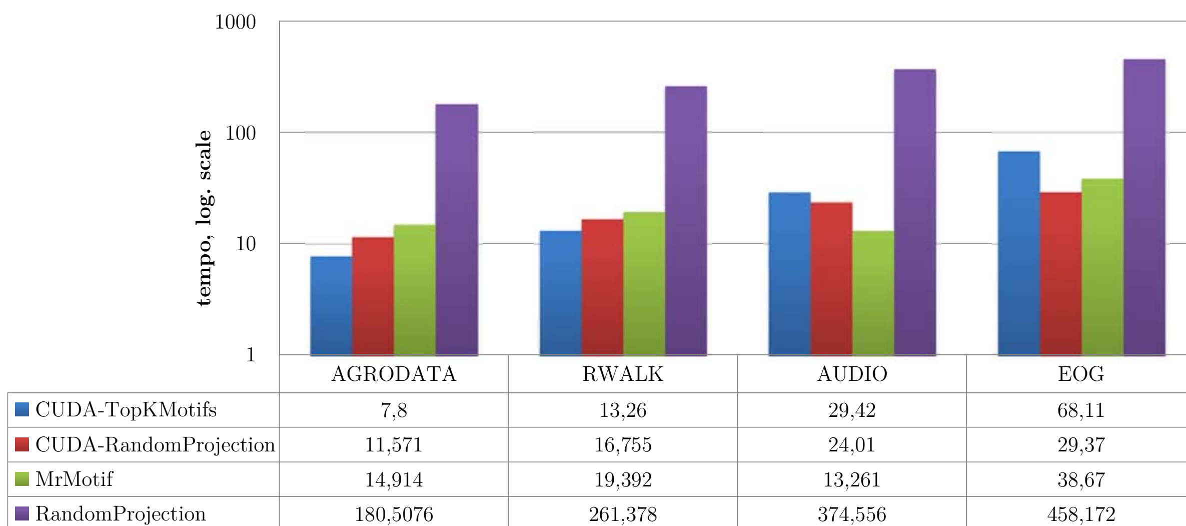
experimentos. Isto demonstra que o método desenvolvido é eficiente e também exato para definições simples dos *motifs*, como do *Pair-Motif*.

Experimento 2: Comparação dos métodos de descoberta dos *TopK-Motifs*

O método CUDA-LSH foi utilizado para melhorar o desempenho de algoritmos para a identificação dos *TopK-Motifs*, ou seja, os K padrões mais frequentes das séries. Desse modo, o algoritmo *CUDA-TopKMotifs()*, proposto na Seção 5.3.1, é avaliado ao se comparar com os métodos de descoberta de *motifs Random Projection*, *CUDA-Random Projection* e *MrMotif*. A importância em se comparar os tempos de processamento na descoberta dos *TopK-Motifs* vem da possiblidade de analisar os comportamentos dos métodos tanto sequenciais quanto paralelas em relação à dimensionalidade e o tamanho do conjunto de dados.

Uma consideração interessante a respeito dos métodos *Random Projection* e *MrMotif* é que ao trabalhar com representações reduzidas das séries originais conseguem reduzir o custo de processamento das séries. Especificamente, esses métodos utilizam o método de representação SAX e iSAX respectivamente. Desse modo os algoritmos *CUDA-RandomProjection* e *MrMotif* são executados com os parâmetros de configuração recomendados por [Lin et al., 2003] e [Castro e Azevedo, 2010]. Assim, o tamanho da palavra SAX foi definido com $w = 8$ para o Algoritmo *CUDA-Random Projection*, e $g_{min} = 4, g_{max} = 64$ como as resoluções mínimas e máximas na representação iSAX para o algoritmo *MrMotif*.

A seguir são descritos os resultado de desempenho dos algoritmos *CUDA-TopKMotifs*, *CUDA-RandomProjection* e *MrMotif*, para a identificação dos *Top-10 Motifs* nos conjuntos de dados AGRODATA, RWALK, AUDIO e EOG. Esses resultados são resumidos nas Figuras 6.8 e 6.9. Note que o parâmetro $TopK$ foi limitado até 10, pois nos experimentos os métodos baseados em representações SAX não conseguiram reportar mais de 10 *motifs* relevantes. Além disso, neste experimento utiliza-se os *clusters* formados pelos 10-vizinhos mais próximos entre si para definir os *motif* relevantes, ou seja, utiliza-se consultas *All 10-NN* como procedimento para definir os *clusters* candidatos.



	AGRODATA	RWALK	AUDIO	EOG
CUDA-TopKMotifs	7,8	13,26	29,42	68,11
CUDA-RandomProjection	11,571	16,755	24,01	29,37
MrMotif	14,914	19,392	13,261	38,67
RandomProjection	180,5076	261,378	374,556	458,172

Figura 6.8: Comparação do tempo de execução para a identificação dos *Top-10 Motifs* utilizando os algoritmos *CUDA-TopKMotifs*, *CUDA-RandomProjection*, *MrMotif* e *Random Projection*.

Uma observação interessante na Figura 6.8 é o custo dos algoritmos conforme a dimensionalidade dos dados cresce. Ao se utilizar as séries originais o algoritmo *CUDA-TopKMotifs* apresenta um custo que cresce conforme a dimensionalidade aumenta. Por exemplo, o algoritmo *CUDA-TopKMotifs* apresenta melhores resultados quando a dimensionalidade é menor do que 32 como pode ser observado para os conjuntos de dados AGRODATA (24-D) e RWALK (32-D). No entanto, como pode ser observado para os conjuntos de dados AUDIO (192-D) e EOG (256-D) quando a dimensionalidade é maior os algoritmos que só processam as palavras SAX conseguem melhores resultados.

Neste experimento, verificou-se a mesma relação de custos apresentado em [Castro e Azevedo, 2010], para os algoritmos sequenciais *Random Projection* e *MrMotif*. Como pode ser observado nas Figuras 6.8 e 6.9 o tempo total gasto para reportar os *Top-10 Motif*, na maioria dos casos, é comparável entre os algoritmos *CUDA-RandomProjection* e *MrMotif*. Isto é revelador, pois significa que um algoritmo sequencial de custo quadrático, como o *Random Projection*, pode ser melhorado significativamente utilizando uma implementação em GPU, neste caso obteve-se um fator de aceleração de até 15x.

Como pode ser observado na Figura 6.9, o comportamento dos algoritmos em relação ao tamanho do conjunto mostra uma relação de crescimento linear até mesmo com o algoritmo *CUDA-RandomProjection*, cuja implementação sequencial em CPU apresenta um custo quadrático. Esses resultados demonstram o potencial da plataforma de programação CUDA para acelerar algoritmo de descoberta de *motifs*.

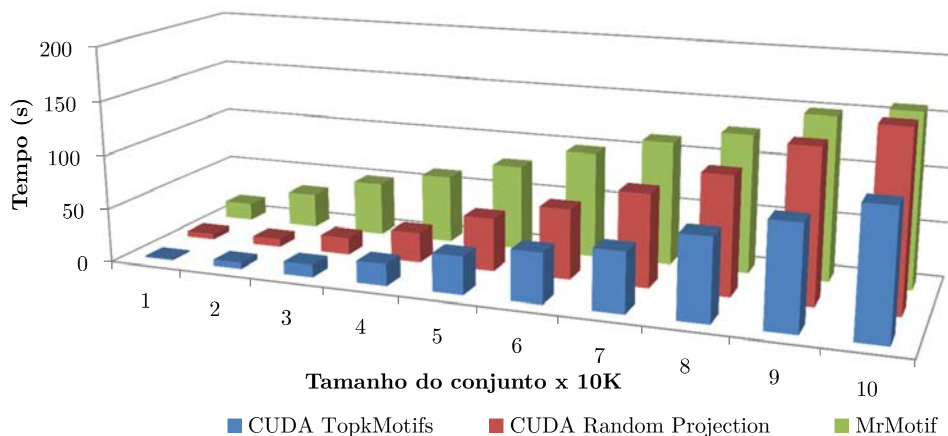


Figura 6.9: Comparação do tempo de execução na identificação dos *Top-10 Motifs* conforme o tamanho do conjunto de dados RWALK cresce.

Finalmente, para avaliar a precisão dos algoritmos de descoberta de *motifs* analisa-se a relevância dos resultados obtidos. Dessa forma, cada *motif* é avaliado pela sua densidade, ou seja, pelo raio do *cluster* que contém os k elementos mais próximos entre si. Como pode se observar na Figura 6.10 os resultados de precisão para o algoritmo *CUDA-TopKMotifs* são melhores, isto é porque tanto o método *Random Projection* como o método *MrMotif* trabalham com dados representações reduzidas dos dados utilizando o método SAX.

Desta forma, foi mostrado que o método proposto é efetivo para a identificação de *motifs* em séries temporais; além disso mostra-se que explorando técnicas aproximadas e programação *multi-threads* em GPU e CPU, a velocidade na procura dos *clusters* por meio de buscas paralelas AllkNN é incrementada, consequentemente, todo o processo de descoberta de *motifs* também é melhorado.

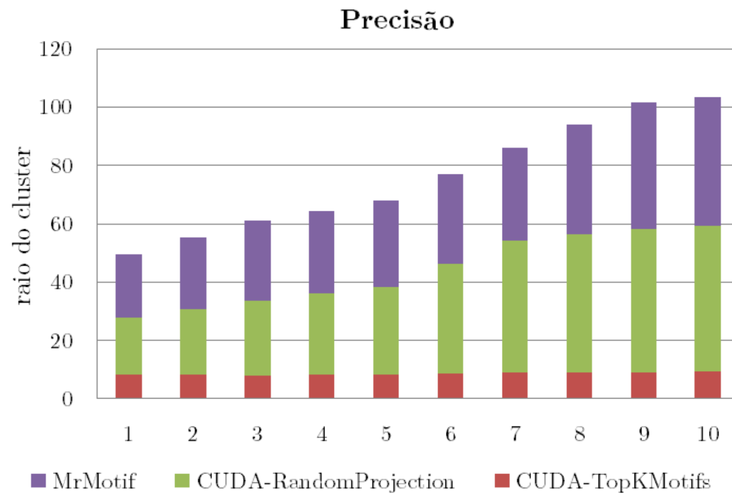


Figura 6.10: Comparação dos raios dos *cluster* para os *Top-10 Motifs* encontrados pelos algoritmos *CUDA-TopKMotifs*, *CUDA-RandomProjection* e *MrMotif*.

6.4 Considerações Finais

Neste capítulo foi realizado um estudo experimental do método proposto e dos respectivos algoritmos, considerando algumas de suas etapas de modo individual. Como discutido anteriormente, a metodologia proposta neste trabalho baseia-se em duas frentes. A primeira, utilizando técnicas eficientes de busca kNN, é amplamente conhecida pela comunidade científica da área de banco de dados; e a segunda, identificação de *motifs*, tem sido cada vez mais estudada e aplicada em distintos domínios. Os estudos experimentais apresentados neste capítulo mostram que tanto a técnica de busca AllkNN, implementada como o algoritmo *CUDA-AllkNN()*, quanto o algoritmo de identificação de *motifs*, implementado como o algoritmo *CUDA-TopKMotifs()*, apresentam resultados de qualidade com tempos de resposta sublineares tanto com conjunto de dados tanto reais como sintéticos.

Os experimentos também mostram que o uso de programação *multi-thread*, garante um maior ganho de desempenho para grandes conjuntos de dados. Portanto, é razoável argumentar que é possível lidar com grandes conjuntos de dados e em altas dimensões com implementações eficientes em arquiteturas multi-núcleos das GPUs. Assim, como o rápido crescimento dos bancos de dados exige soluções de alto desempenho mais eficientes de busca kNN e descoberta de *motifs*, os resultados apresentados neste trabalho são relevantes.

CAPÍTULO

7

Conclusão

7.1 Considerações Finais

A crescente disponibilidade de dados em diferentes domínios tem motivado o desenvolvimento de técnicas e métodos capazes de descobrir conhecimento em grandes volumes de dados complexos. Em domínios complexos, aplicações das áreas de recuperação de informação e mineração de dados precisam de algoritmos eficientes e eficazes. Muitas dessas soluções fornecem resultados exatos, mas apesar das recentes pesquisas em algoritmos exatos, os algoritmos aproximados vêm se mostrando uma opção melhor em diversos domínios de aplicação, devido a sua eficiência em tempo e espaço, especialmente nos contextos em que a velocidade de resposta é mais importante que a precisão dos resultados.

Na área de recuperação de informação, a busca por similaridade em dados complexos é um campo de pesquisa importante, já que muitas tarefas de mineração de dados, como a classificação, detecção de agrupamentos e descoberta de *motifs*, dependem de algoritmos de busca ao vizinho mais próximo. No entanto, mesmo considerando soluções aproximadas como uma abordagem para acelerar consultas por similaridade, muitas soluções simples e gerais estão sendo consideradas impraticáveis em contextos em que o alto custo computacional é um limitante.

Para melhorar os tempos de execução de operações de busca e das tarefas de mineração de dados, demonstrou-se na literatura que desenvolvendo soluções de alto desempenho utilizando o paradigma de programação GPGPU consegue-se melhorar o seu desempenho em várias ordens de magnitude.

Neste trabalho foram propostos algoritmos de alto desempenho para descoberta de *motifs* que utilizam a abordagem de busca por similaridade aproximada e os recursos do paradigma de programação GPGPU. Para fornecer buscas por similaridade de alto desempenho foi desenvolvido a método de indexação de dados multidimensionais CUDA-LSH. Desse modo, utilizando implementações eficientes e paralelas de buscas AllkNN, foi proposto um algoritmo paralelo de descoberta de *motifs* baseado na definição dos K padrões mais frequentes. Para avaliar os resultados, diversos experimentos foram realizados com dados reais e sintéticos, além de comparação entre implementações distintas em CPU e

GPU.

Em resumo, três aspectos desta pesquisa devem ser destacados. O primeiro foi em relação ao estudo de algoritmos de busca por similaridade, focado em algoritmos aproximados. O segundo aspecto foi a adaptação dos algoritmos para arquiteturas paralelas em CPU e GPU. O terceiro foi o estudo da importância dos algoritmos aproximados associada à computação GPGPU na identificação de *motifs*.

Em relação aos algoritmos de busca por similaridade, a avaliação foi realizada com a seguinte metodologia: avaliação dos algoritmos de busca por abrangência exata e aproximada, focado nos métodos baseados no esquema LSH; avaliação dos algoritmos aproximados, com o intuito de possibilitar a identificação do método mais apropriado para fornecer buscas kNN aproximadas, para que pudesse ser implementado em CUDA.

Nas duas avaliações, os métodos LSH, mostraram um melhor desempenho em relação aos outros algoritmos exatos. No entanto, devido às restrições de implementação da plataforma de programação CUDA, foi necessária a adaptação e redefinição das operações de construção e consulta, assim como a simplificação das estruturas de dados para serem projetadas em GPU.

Desse modo, o método CUDA-LSH foi desenvolvido para explorar as técnicas aproximadas e os recursos de computação de alto desempenho fornecidos pela GPU. mostrou-se que para poder atingir altos níveis de desempenho, um esquema de indexação deve ser projetado de tal forma que a maioria de etapas de computação sejam executadas na GPU, para poder minimizar o fluxo de dados entre a CPU e a GPU. Este foi o critério de projeto mais marcante da técnica desenvolvida.

Uma vez definido o método de busca para consultas de alto desempenho, o método CUDA-LSH foi projetada para fornecer consultas paralelas AllkNN. Para questões de avaliação duas implementações, o *Par-AllkNN()* e o *CUDA-AllkNN()*, foram desenvolvidas e avaliadas comparando-se os tempos de processamento em diferentes níveis de paralelismo e configurações de dados. Nessa comparação, o algoritmo *CUDA-AllkNN()* demonstrou o potencial da plataforma de programação CUDA, apresentando uma aceleração de até 7 (sete) vezes ao se comparar com a versão em CPU. Portanto, mostrou-se que é possível utilizar a aceleração por hardware fornecido pela GPU para consultas paralelas AllkNN.

Em relação aos algoritmos para descoberta de *motifs*, apresentou-se uma avaliação tanto quantitativa como qualitativa para as distintas soluções exatas e aproximadas. Assim, a avaliação foi dividia em duas partes: avaliação dos algoritmos exatos e aproximados para a identificação do *Pair-Motif*; e, finalmente, uma avaliação do método proposto, o algoritmo *CUDA-TopKMotifs()*, com os outros algoritmos probabilísticos e aproximados de identificação de *motifs*.

Os resultados experimentais mostram que para o método de descoberta de *motifs*, implementado pelo Algoritmo *CUDA-TopKMotifs()*, ao se explorar soluções aproximadas de busca e as capacidades de computação de uso geral das GPUs consegue-se melhorar o desempenho dos algoritmos. Esses resultados mostram também que o equilíbrio entre o desempenho e a precisão dos algoritmos que utilizam as buscas kNN como o procedimento mais importante é garantido devido à utilização de técnicas aproximadas de busca que permitem consultas de custo sublinear para dados em altas dimensões.

De modo geral, os resultados deste estudo mostraram que a exploração adequada de técnicas aproximadas e programação *multi-thread* em problemas complexos de recuperação e mineração de dados garante um ganho significativo de desempenho, especialmente para algoritmos que demandem grandes recursos de computação.

7.2 Principais Contribuições

Este trabalho de mestrado explorou, a aplicação de métodos de busca por similaridade aproximada e computação GPGPU na identificação de *motifs* em series temporais. Além disso, estudos iniciais indicam resultados promissores na utilização da técnica desenvolvida, o CUDA-LSH, em outras tarefas de mineração de series temporais que precisam que precisam de grandes recursos de computação para calcular a distancia entre series.

As principais contribuições deste trabalho são sintetizadas nos itens seguintes:

1. Desenvolvimento do método de indexação de dados multidimensionais CUDA-LSH para buscas por similaridade de alto desempenho, mostrou-se que para poder atingir altos níveis de desempenho, um esquema de indexação em GPU deve ser projetado de tal forma que a maioria de etapas de computação sejam executadas na GPU;

2. Desenvolvimento do algoritmo *CUDA-AllkNN()* para buscas paralelas a todos os k-vizinhos mais próximos utilizando a plataforma de programação CUDA e o método CUDA-LSH, os resultados mostram uma aceleração em consultas por similaridade de até sete vezes ao se comprar com a versão em CPU;

3. Desenvolvimento do algoritmo *CUDA-TopKMotifs()* para a identificação dos K padrões mais frequentes baseado no algoritmo *CUDA-AllkNN()*, os resultados mostram uma aceleração de até sete vezes ao se comparar com a versão em CPU do algoritmo para a identificação do *Pair-Motif* e resultados comparáveis com o algoritmo *CUDA-RandomProjection* e *MrMotif* no tempo de execução dos algoritmos mas com melhores resultados de precisão;

4. Desenvolvimento do algoritmo *CUDA-RandomProjection()* para a identificação dos K padrões mais frequentes baseado em *Random Projection*, os resultados mostram uma aceleração de até quinze vezes ao se comparar com a versão em CPU do algoritmo *RandomProjection*;

5. Análise e importância da computação GPGPU e das abordagens aproximadas para busca por similaridade em tarefas de mineração de dados. Os resultados mostraram que o uso de programação *multi-thread*, especificamente programação em GPGPU, garante um maior ganho de desempenho para grandes conjuntos de dados e altas dimensões tanto para problemas de busca por similaridade como para problemas de identificação de *motifs*.

Estas contribuições resultaram em:

Alexander Ocsa and Elaine P. M. Sousa (2010). An Adaptive Multi-level Hashing Structure for Fast Approximate Similarity Search. Journal of Information and Data Management (JIDM), SBBD Edition, 1(3):359-374.

7.3 Propostas de Trabalhos Futuros

O trabalho de desenvolvimento do algoritmo *CUDA-AllkNN()* abriu ainda a possibilidade de novas frentes de pesquisa, entre as quais encontram-se:

1. **Melhoramento do algoritmo *CUDA-AllkNN()*:** Os experimentos mostraram que mesmo usando arquiteturas de programação paralela, a alta dimensionalidade dos dados ainda é uma limitante no desempenho dos algoritmos. Por exemplo, em alguns casos onde a dimensionalidade dos dados é muito alta, algoritmos baseados em representações SAX e iSAX mostraram tempos de execução melhores. Assim, pode-se reduzir o custo computacional utilizando a representação SAX ou iSAX, trabalhando desta forma apenas com os dados reduzidos.

2. **Extensão do algoritmo *CUDA-AllkNN()* para outras tarefas de mineração de dados:** O algoritmo *CUDA-AllkNN()* mostrou-se útil para acelerar algoritmos de descoberta de *motifs*. No entanto, os algoritmos de descoberta de *motifs* não são os únicos algoritmos de mineração de dados que dependem do algoritmo de busca kNN. Especificamente, tarefas de mineração de dados que têm a busca AllkNN como procedimento principal, como a detecção de agrupamentos, classificação e detecção de intrusos, podem ser melhoradas usando o algoritmo *CUDA-AllkNN()*.

Referências Bibliográficas

[Agrawal et al., 1993] Agrawal, R., Faloutsos, C., e Swami, A. N. (1993). Efficient Similarity Search In Sequence Databases. In *Proceedings of the 4th International Conference on Foundations of Data Organization and Algorithms*, páginas 69–84, London, UK.

[Andoni e Indyk, 2008] Andoni, A. e Indyk, P. (2008). Near-optimal hashing algorithms for approximate nearest neighbor in high dimensions. *Communications of the ACM*, 51(1):117–122.

[Assent et al., 2008] Assent, I., Krieger, R., Afschari, F., e Seidl, T. (2008). The TS-tree: efficient time series search and retrieval. In *Proceedings of the International Conference on Extending Database Technology*, páginas 252–263, New York, NY, USA.

[Bailey e Elkan, 1995] Bailey, T. L. e Elkan, C. (1995). Unsupervised learning of multiple motifs in biopolymers using expectation maximization. *Machine Learning*, 21:51–80.

[Bawa et al., 2005] Bawa, M., Condie, T., e Ganesan, P. (2005). LSH forest: self-tuning indexes for similarity search. In *Proceedings of the 14th international conference on World Wide Web*, páginas 651–660, Chiba, Japan.

[Beckmann et al., 1990] Beckmann, N., Kriegel, H.-P., Schneider, R., e Seeger, B. (1990). The R*-tree: an efficient and robust access method for points and rectangles. *SIGMOD Record*, 19(2):322–331.

[Bellman, 1961] Bellman, R. E. (1961). *Adaptive control processes - A guided tour*, volume 1961, chapter (A RAND Corporation Research Study), páginas 255–260. Princeton University Press.

[Bentley, 1979] Bentley, J. L. (1979). Multidimensional Binary Search Trees in Database Applications. *Transactions on Software Engineering*, 5(4):333–340.

[Berchtold et al., 1996] Berchtold, S., Keim, D. A., e Kriegel, H.-P. (1996). The X-tree: An Index Structure for High-Dimensional Data. In *Proceedings of the International Conference on Very Large Data Bases*, páginas 28–39, San Francisco, CA, USA.

[Berndt e Clifford, 1994] Berndt, D. J. e Clifford, J. (1994). Using Dynamic Time Warping to Find Patterns in Time Series. In *Proceedings of KDD: AAAI Workshop on Knowledge Discovery in Databases*, páginas 359–370, Seattle, Washington, USA.

[Blott e Weber, 2008] Blott, S. e Weber, R. (2008). What's wrong with high-dimensional similarity search. In *Proceedings of the International Conference on Very Large Data Bases*, páginas 3–3, Aukland, New Zealand.

[Böhm et al., 2001] Böhm, C., Berchtold, S., e Keim, D. A. (2001). Searching in high-dimensional spaces: Index structures for improving the performance of multimedia databases. *ACM Computing Surveys*, 33(3):322–373.

[Böhm et al., 2009] Böhm, C., Noll, R., Plant, C., Wackersreuther, B., e Zherdin, A. (2009). *Transactions on Large-Scale Data- and Knowledge-Centered Systems I*, volume 5740, chapter Data Mining Using Graphics Processing Units, páginas 63–90. Springer-Verlag, Berlin, Heidelberg.

[Buhler e Tompa, 2001] Buhler, J. e Tompa, M. (2001). Finding motifs using random projections. In *Proceedings of the annual international conference on Computational biology*, páginas 69–76, New York, NY, USA.

[Castro e Azevedo, 2010] Castro, N. e Azevedo, P. (2010). Multiresolution Motif Discovery in Time Series. In *Proceedings of the SIAM International Conference on Data Mining*, páginas 665–676, Columbus, Ohio, USA.

[Chan e Fu, 1999] Chan, K. e Fu, A. W. (1999). Efficient Time Series Matching by Wavelets. In *Proceedings of the IEEE International Conference on Data Engineering*, páginas 126–130, Washington, DC, USA.

[Charles et al., 2008] Charles, J., Potok, T., Patton, R., e Cui, X. (2008). Flocking-based Document Clustering on the Graphics Processing Unit. In *Proceedings of the International Workshop on Nature Inspired Cooperative Strategies for Optimization*, volume 129, páginas 27–37.

[Chávez et al., 2001] Chávez, E., Navarro, G., Baeza-Yates, R., e Marroquín, J. L. (2001). Searching in metric spaces. *ACM Computing Surveys*, 33(3):273–321.

[Che et al., 2008] Che, S., Boyer, M., Meng, J., Tarjan, D., Sheaffer, J. W., e Skadron, K. (2008). A performance study of general-purpose applications on graphics processors using CUDA. *Journal of Parallel and Distributed Computing*, 68:1370–1380.

[Chiu et al., 2003] Chiu, B., Keogh, E., e Lonardi, S. (2003). Probabilistic discovery of time series motifs. In *Proceedings of the ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, páginas 493–498, New York, NY, USA.

[Ciaccia et al., 1997] Ciaccia, P., Patella, M., e Zezula, P. (1997). M-tree: An Efficient Access Method for Similarity Search in Metric Spaces. In *Proceedings of the International Conference on Very Large Data Bases*, páginas 426–435, San Francisco, CA, USA.

[Clarkson, 2006] Clarkson, K. L. (2006). *Nearest-Neighbor Methods for Learning and Vision: Theory and Practice*, chapter Nearest-Neighbor Searching and Metric Space Dimensions, páginas 15–59. MIT Press.

[Datar et al., 2004] Datar, M., Immorlica, N., Indyk, P., e Mirrokni, V. S. (2004). Locality-sensitive hashing scheme based on p-stable distributions. In *Proceedings of the annual symposium on computational geometry*, páginas 253–262, New York, NY, USA.

[Ding et al., 2008] Ding, H., Trajcevski, G., Scheuermann, P., Wang, X., e Keogh, E. (2008). Querying and mining of time series data: experimental comparison of representations and distance measures. *Proceedings of the VLDB Endowment*, 1(2):1542–1552.

[Dong et al., 2008] Dong, W., Wang, Z., Josephson, W., Charikar, M., e Li, K. (2008). Modeling LSH for performance tuning. In *Proceedings of the International Conference on Information and Knowledge Engineering*, páginas 669–678, Napa Valley, California, USA.

[Faloutsos, 1986] Faloutsos, C. (1986). Multiattribute hashing using Gray codes. *SIGMOD Record*, 15(2):227–238.

[Faloutsos et al., 1994] Faloutsos, C., Ranganathan, M., e Manolopoulos, Y. (1994). Fast subsequence matching in time-series databases. In *Proceedings of the ACM SIGMOD International Conference on Management of Data Conference*, páginas 419–429, New York, NY, USA.

[Faloutsos e Roseman, 1989] Faloutsos, C. e Roseman, S. (1989). Fractals for secondary key retrieval. In *Proceedings of the ACM Symposium on Principles of Database Systems*, páginas 247–252, New York, NY, USA.

[Fayyad et al., 1996] Fayyad, U. M., Piatetsky-Shapiro, G., e Smyth, P. (1996). From data mining to knowledge discovery: an overview. In *Advances in knowledge discovery and data mining*, páginas 1–34. American Association for Artificial Intelligence.

[Fu et al., 2008] Fu, A. W.-C., Keogh, E., Lau, L. Y., Ratanamahatana, C. A., e Wong, R. C.-W. (2008). Scaling and time warping in time series querying. *The VLDB Journal*, 17(4):899–921.

[Gaede e Gunther, 1998] Gaede, V. e Gunther, O. (1998). Multidimensional Access Methods. *ACM Computing Surveys*, 30(2):170–231.

[Garcia et al., 2008] Garcia, V., Debreuve, E., e Barlaud, M. (2008). Fast k nearest neighbor search using GPU. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, páginas 1 –6, Anchorage, UK, USA.

[Gionis et al., 1999] Gionis, A., Indyk, P., e Motwani, R. (1999). Similarity Search in High Dimensions via Hashing. In *Proceedings of the International Conference on Very Large Data Bases*, páginas 518–529, San Francisco, CA, USA.

[Govindaraju et al., 2006] Govindaraju, N., Gray, J., Kumar, R., e Manocha, D. (2006). GPUTeraSort: high performance graphics co-processor sorting for large database management. In *Proceedings of the ACM SIGMOD International Conference on Management of Data Conference*, páginas 325–336, New York, NY, USA.

[Govindaraju et al., 2004] Govindaraju, N. K., Lloyd, B., 0010, W. W., Lin, M. C., e Manocha, D. (2004). Fast computation of database operations using graphics processors. In *Proceedings of the ACM SIGMOD International Conference on Management of Data Conference*, páginas 215–226, New York, NY, USA.

[Guttman, 1984] Guttman, A. (1984). R-trees: a dynamic index structure for spatial searching. *SIGMOD Record*, 14:47–57.

[Hisashi Koga, 2004] Hisashi Koga, Tetsuo Ishibashi, T. W. (2004). Fast Hierarchical Clustering Algorithm Using Locality-Sensitive Hashing. *Discovery Science*, 12(1):155–182.

[Hisashi Koga, 2007] Hisashi Koga, Tetsuo Ishibashi, T. W. (2007). Fast agglomerative hierarchical clustering algorithm using Locality-Sensitive Hashing. *Knowledge and Information Systems*, 12(1):25–53.

[Hjaltason e Samet, 2003] Hjaltason, G. R. e Samet, H. (2003). Index-driven similarity search in metric spaces. *ACM Transactions on Database Systems*, 28(4):517–580.

[Indyk e Motwani, 1998] Indyk, P. e Motwani, R. (1998). Approximate nearest neighbors: towards removing the curse of dimensionality. In *Proceedings of the thirtieth annual ACM symposium on Theory of computing*, páginas 604–613, New York, NY, USA.

[Keogh, 2002] Keogh, E. (2002). Exact indexing of dynamic time warping. In *Proceedings of the International Conference on Very Large Data Bases*, páginas 406–417, Hong Kong, China.

[Keogh, 2003] Keogh, E. (2003). Data Mining and Machine Learning in Time Series Databases (Tutorial).

[Keogh et al., 2001] Keogh, E., Chakrabarti, K., Pazzani, M., e Mehrotra, S. (2001). Locally adaptive dimensionality reduction for indexing large time series databases. *SIGMOD Record*, 30(2):151–162.

[Kirk e Hwu, 2010] Kirk e Hwu, W. (2010). *Programming Massively Parallel Processors: A Hands-on Approach*, chapter CUDA Threads, páginas 59–74. Elsevier Inc.

[Korn et al., 1997] Korn, F., Jagadish, H. V., e Faloutsos, C. (1997). Efficiently supporting ad hoc queries in large datasets of time sequences. *SIGMOD Record*, 26(2):289–300.

[Kulis e Grauman, 2009] Kulis, B. e Grauman, K. (2009). Kernelized locality-sensitive hashing for scalable image search. In *Proceedings of the International Conference on Computer Vision*, páginas 2130 –2137, Kyoto, Japan.

[Liang et al., 2010] Liang, S., Liu, Y., Wang, C., e Jian, L. (2010). Design and evaluation of a parallel k-nearest neighbor algorithm on CUDA-enabled GPU. In *Proceedings of IEEE 2nd Symposium on Web Society*, páginas 53–60, Beijing, China.

[Liao et al., 2001] Liao, S., Lopez, M. A., e Leutenegger, S. T. (2001). High Dimensional Similarity Search With Space Filling Curves. In *Proceedings of the IEEE International Conference on Data Engineering*, páginas 615–622, Washington, DC, USA.

[Lieberman et al., 2008] Lieberman, M., Sankaranarayanan, J., e Samet, H. (2008). A Fast Similarity Join Algorithm Using Graphics Processing Units. In *Proceedings of the IEEE International Conference on Data Engineering*, páginas 1111–1120, Cancún, Mexico.

[Lin et al., 2003] Lin, J., Keogh, E., Lonardi, S., e Chiu, B. (2003). A symbolic representation of time series, with implications for streaming algorithms. In *Proceedings of the ACM SIGMOD workshop on Research issues in data mining and knowledge discovery*, páginas 2–11, New York, NY, USA.

[Lin et al., 2007] Lin, J., Keogh, E., Wei, L., e Lonardi, S. (2007). Experiencing SAX: a novel symbolic representation of time series. *Data Mining and Knowledge Discovery*, 15(2):107–144.

[Lin et al., 2002] Lin, J., Keogh, E. J., Lonardi, S., e Patel, P. (2002). Finding Motifs in Time Series. In *Proceedings of the International Conference on Knowledge Discovery and Data Mining*, páginas 53–68, Edmonton, Alberta, Canada.

[Lindholm et al., 2008] Lindholm, E., Nickolls, J., Oberman, S., e Montrym, J. (2008). Nvidia tesla: A unified graphics and computing architecture. *Micro, IEEE*, 28:39 –55.

[Liu et al., 2006] Liu, T., Moore, A. W., e Gray, A. (2006). New algorithms for efficient high-dimensional nonparametric classification. *Journal of Machine Learning Research*, 7:1135–1158.

[Liu et al., 2004] Liu, T., Moore, A. W., Gray, A., e Yang, K. (2004). An investigation of practical approximate nearest neighbor algorithms. In *In proceedings of Neural Information Processing Systems*, páginas 825–832, Vancouver, BC, Canada.

[Liu et al., 2010] Liu, Y., Schmidt, B., Liu, W., e Maskell, D. L. (2010). CUDA-MEME: Accelerating motif discovery in biological sequences using CUDA-enabled graphics processing units. *Pattern Recognition Letters*, 31:2170–2177.

[Lv et al., 2007] Lv, Q., Josephson, W., Wang, Z., Charikar, M., e Li, K. (2007). Multi-probe LSH: efficient indexing for high-dimensional similarity search. In *Proceedings of the International Conference on Very Large Data Bases*, páginas 950–961, Vienna, Austria.

[Metwally et al., 2005] Metwally, A., Agrawal, D., e Abbadi, A. E. (2005). Efficient computation of frequent and top-k elements in data streams. In *Proceedings of the International Conference on Database Theory*, páginas 398–412, Edinburgh, Scotland.

[Mueen et al., 2009] Mueen, A., Keogh, E. J., Zhu, Q., Cash, S., e Westover, B. (2009). Exact Discovery of Time Series Motifs. In *Proceedings of the IEEE International Conference on Data Mining*, páginas 473–484, Sparks, NV, USA.

[Navarro, 2002] Navarro, G. (2002). Searching in metric spaces by spatial approximation. *The VLDB Journal*, 11(1):28–46.

[Navarro et al., 2007] Navarro, G., Paredes, R., e Chávez, E. (2007). t-Spanners for metric space searching. *Data and Knowledge Engineering*, 63:820–854.

[Nickolls et al., 2008] Nickolls, J., Buck, I., Garland, M., e Skadron, K. (2008). Scalable Parallel Programming with CUDA. *Queue*, 6:40–53.

[Ocsa et al., 2007] Ocsa, A., Bedregal, C., e Cuadros-Vargas, E. (2007). A new approach for similarity queries using neighborhood graphs. In *Proceedings of the Brazilian Symposium on Databases*, páginas 131–142, João Pessoa, Paraíba, Brasil.

[Ocsa e Sousa, 2010] Ocsa, A. e Sousa, E. P. M. (2010). An Adaptive Multi-level Hashing Structure for Fast Approximate Similarity Search. *Journal of Information and Data Management*, 1(3):359–374.

[Owens J. e Purcell, 2007] Owens J., L. D. G. N. H. M. K. L. A. e Purcell (2007). A Survey of General-Purpose Computation on Graphics Hardware. *Computer Graphics Forum*, 26(1):80–113.

[Sagan, 1994] Sagan, H. (1994). *Space-Filling Curves*. Springer-Verlag, New York.

[Sellis et al., 1987] Sellis, T. K., Roussopoulos, N., e Faloutsos, C. (1987). The R+-Tree: A Dynamic Index for Multi-Dimensional Objects. In *Proceedings of the International Conference on Very Large Data Bases*, páginas 507–518, San Francisco, CA, USA.

[Shepherd et al., 1999] Shepherd, J., Zhu, X., e Megiddo, N. (1999). A Fast Indexing Method for Multidimensional Nearest Neighbor Search. In *Proceedings of SPIE - Conference on Storage and Retrieval for Image and Video Databases*, páginas 350–355, San Jose, California, USA.

[Shieh e Keogh, 2008] Shieh, J. e Keogh, E. (2008). iSAX: indexing and mining terabyte sized time series. In *Proceedings of the ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, páginas 623–631, New York, NY, USA.

[Skopal et al., 2005] Skopal, T., Pokorný, J., e Snásel, V. (2005). Nearest Neighbours Search Using the PM-Tree. In *In Proceedings of Database Systems for Advanced Applications*, páginas 803–815, Beijing, China.

[Slaney e Casey, 2008] Slaney, M. e Casey, M. (2008). Locality-Sensitive Hashing for Finding Nearest Neighbors. *IEEE Signal Processing Magazine*, 25(1):128–131.

[Stone et al., 2010] Stone, J., Gohara, D., e Shi, G. (2010). OpenCL: A Parallel Programming Standard for Heterogeneous Computing Systems. *Computing in Science Engineering*, 12:66 –73.

[Tan et al., 2005] Tan, P.-N., Steinbach, M., e Kumar, V. (2005). *Introduction to Data Mining, (First Edition)*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA.

[Tao et al., 2010] Tao, Y., Yi, K., Sheng, C., e Kalnis, P. (2010). Efficient and accurate nearest neighbor and closest pair search in high-dimensional space. *ACM Transactions on Database Systems*, 35(3):1–46.

[Traina et al., 2002] Traina, Jr., C., Traina, A., Filho, R. S., e Faloutsos, C. (2002). How to improve the pruning ability of dynamic metric access methods. In *Proceedings of the International Conference on Information and Knowledge Engineering*, páginas 219–226, New York, NY, USA.

[Traina C. et al., 2002] Traina C., J., Traina, A., Faloutsos, C., e Seeger, B. (2002). Fast Indexing and Visualization of Metric Data Sets using Slim-Trees. *IEEE Transactions on Knowledge and Data Engineering*, 14(2):244–260.

[Valle, 2008] Valle, E. (2008). *Local descriptor matching for image identification systems*. Phd thesis, Université de Cergy-Pontoise, France.

[Vieira et al., 2004] Vieira, M. R., Jr., C. T., Chino, F. J. T., e Traina, A. J. M. (2004). DBM-Tree: A Dynamic Metric Access Method Sensitive to Local Density Data. In *Proceedings of the Brazilian Symposium on Databases*, páginas 163–177, Brasília, Brasil.

[Vlachos et al., 2006] Vlachos, M., Hadjieleftheriou, M., Gunopulos, D., e Keogh, E. (2006). Indexing Multidimensional Time-Series. *The VLDB Journal*, 15(1):1–20.

[Volnyansky e Pestov, 2009] Volnyansky, I. e Pestov, V. (2009). Curse of Dimensionality in Pivot Based Indexes. In *Proceedings of the International Workshop on Similarity Search and Applications*, Washington, DC, USA.

[Wang et al., 2010] Wang, J., Kumar, S., e Chang, S.-F. (2010). Semi-Supervised Hashing for Scalable Image Retrieval. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, páginas 3424 –3431, San Francisco, USA.

[Wu et al., 2009] Wu, R., Zhang, B., e Hsu, M. (2009). Clustering billions of data points using GPUs. In *Proceedings of the combined workshops on UnConventional high performance computing workshop plus memory access workshop*, páginas 1–6, New York, NY, USA.

[Yianilos, 1993] Yianilos, P. N. (1993). Data structures and algorithms for nearest neighbor search in general metric spaces. In *Proceedings - Annual Symposium on Discrete Algorithms*, páginas 311–321, Philadelphia, PA, USA.

[Zhang et al., 2011] Zhang, D., Agrawal, D., Chen, G., e Tung, A. (2011). HashFile: An efficient index structure for multimedia data. In *Proceedings of the IEEE International Conference on Data Engineering*, páginas 1103 –1114, Hannover, Germany.

[Zhou et al., 2008] Zhou, K., Hou, Q., Wang, R., e Guo, B. (2008). Real-time kd-tree construction on graphics hardware. *ACM Transactions on Graphics*, 27(5):126:1–126:11.