Universidade Federal de Pernambuco
Centro de Informática

Graduação em Ciência da Computação

Implementação de um Mapa Auto-Organizável em GPU

Mário Henrique Santos da Silva

Trabalho de Graduação

Recife
Julho de 2017

Universidade Federal de Pernambuco
Centro de Informática

Mário Henrique Santos da Silva

Implementação de um Mapa Auto-Organizável em GPU

Trabalho apresentado ao Programa de Graduação em Ciência da Computação do Centro de Informática da Universidade Federal de Pernambuco como requisito parcial para obtenção do grau de Bacharel em Ciência da Computação.

Orientador: *Prof. Hansenclever Bassani*

Recife
Julho de 2017

To W.W. My Star, My Perfect Silence.

Agradecimentos

Gostaria de agradecer ao professor Hansenclever Bassani pela confiança e orientação neste trabalho, e aos outros alunos dele, Pedro, Eduardo, Felipe e João, pela ajuda prestada e conhecimento compartilhado.

Aos professores do Centro de Informática (CIn) da Universidade Federal de Pernambuco e seus respectivos monitores, por todo o aprendizado ao longo desses 6 anos. Aos funcionários e coordenadores do centro, que junto com os professores, fazem do CIn um centro de excelência em computação no Brasil. E aos amigos e colegas de turma com quem eu estudei, e que, ocasionalmente, me carregaram nos projetos.

Aos coordenadores e colegas de trabalho dos meus estágios na Suati, Google e In Loco Media, pela experiência e aprendizado.

Aos membros do grupo da Maratona de Programação do CIn, projeto no qual eu aprendi e cresci muito nesses anos, e que me abriu muitas oportunidades, principalmente pra viajar. Agradeço especialmente aos meus companheiros de time, Gustavo Stor, Lucas Lima e Duhan Caraciolo. Aos competidores veteranos do grupo e ao professor Pedro Manhães, pelo legado de conhecimento e experiência. Às treinadoras Katia Guimarães e Liliane Salgado, responsáveis pelo sucesso não só dos times em que participei, mas também de todos os outros ao longo desses anos. Aos amigos e adversários em outras universidades pelo Brasil, principalmente os que conheci na Escola de Verão da Unicamp. Aos responsáveis pela organização da Maratona de Programação no Brasil e do ACM ICPC, que tornaram tudo isso possível. E também aos responsáveis por *Online Judges* e competições *online* de programação, pela diversão, aprendizado e camisetas.

A todos do mundo que trabalham para criar e disponibilizar os softwares, ferramentas e bibliotecas gratuitos que utilizei ao longo do curso e da vida. A todos que trabalham para compartilhar conhecimento em tutoriais, artigos, ou repositórios como a Wikipédia e Stack Overflow.

E finalmente à minha família, os meus pais Ivânia e Mário Augusto e meus irmãos Vinícius e Narayna, por todo o amor, carinho e apoio. Sem vocês eu não seria nada.

Porque, meu bem, ninguem é perfeito e a vida é assim...
—CONDE

Resumo

À medida que aplicações computacionais geram um volume cada vez maior de dados, aumenta a demanda por algoritmos de aprendizagem de máquina para processar e extrair conhecimento desses dados. Para tornar possível a utilização em larga escala desses algoritmos, são utilizadas diversas técnicas para melhorar sua eficiência. Uma das mais importantes recentemente é o uso de processamento paralelo de GPUs (Unidades de Processamento Gráfico). Para obter ganhos de desempenho, um algoritmo precisa ter sua implementação adaptada para tirar proveito da execução de múltiplas operações em paralelo. Atualmente, tal técnica tem acelerado significativamente o treinamento de redes neurais artificiais de aprendizagem profunda, mas também poderia trazer benefícios para algoritmos não-supervisionados, como mapas auto-organizáveis. Este trabalho tem o objetivo de reimplementar um algoritmo de agrupamento de dados utilizando processamento paralelo em GPU, e avaliar sob que condições se obtém algum ganho de desempenho.

Palavras-chave: aprendizagem de máquina, agrupamento, agrupamento em subespaço, mapas auto-organizáveis, larfdssom, gpu, computação paralela, torch, otimização

Sumário

1 Introdução **1**
- 1.1 Contexto e Motivação 1
- 1.2 Objetivos 2
- 1.3 Estrutura do Trabalho 2

2 Agrupamento em Subespaços com Mapas Auto-Organizáveis **3**
- 2.1 Mapa Auto-Organizável 3
- 2.2 Mapa Auto-Organizável Seletivo de Dimensões (DSSOM) 4
- 2.3 LARFDSSOM 5

3 Implementação do LARFDSSOM em GPU **10**
- 3.1 Programação com GPUs 10
- 3.2 Linguagem de programação e *Framework* 10
- 3.3 Exemplo de uso de tensores 11
- 3.4 Implementação do LARFDSSOM usando tensores 13
  - 3.4.1 Variáveis 13
  - 3.4.2 Treinamento 14
    - 3.4.2.1 Competição 16
    - 3.4.2.2 Inserção de Neurônios 16
    - 3.4.2.3 Atualização do Mapa 16
    - 3.4.2.4 Remoção de Neurônios 17
  - 3.4.3 Agrupamento Final 17

4 Testes **19**
- 4.1 Configurações dos Testes 19
- 4.2 Testes com Bases de Dados do OpenSubspace 20
  - 4.2.1 Qualidade do Agrupamento 21
  - 4.2.2 Tempo de Execução 21
- 4.3 Testes com Bases de Dados Adicionais 24
  - 4.3.1 Qualidade do Agrupamento 24
  - 4.3.2 Tempo de Execução 25

5 Conclusão **29**

Lista de Figuras

2.1 Estrutura de um Mapa Auto-Organizável. 4

2.2 Representação do treinamento de um Mapa Auto-Organizável. A massa azul representa o conjunto de padrões de entrada e o ponto branco é o padrão escolhido na iteração atual. Já a grade preta são os vetores de pesos e suas conexões, e o ponto destacado em amarelo é o vetor do vencedor. Ele e seus vizinhos são então movidos para mais próximo do padrão de entrada. 5

3.1 Operações comuns com tensores no *framework* Torch: (a) *Reduce*: calcular uma função que agrega todos os elementos do tensor, como soma, máximo ou mínimo; (b) Map: aplicar uma função independentemente para cada elemento, como multiplicação por um escalar ou elevar ao quadrado; (c) Zip e Map: calcular uma função elemento-a-elemento entre dois tensores, como soma ou subtração. 11

3.2 Ilustração dos tensores usados para armazenar as variáveis associadas aos neurônios. Os tensores dos contadores de vitórias (*Wins*) e das normas dos vetores de relevância (*Relevance_norms*) têm o formato do tensor unidimensional à esquerda. Os tensores *Weights*, *Distances* e *Relevances* têm o formato do segundo tensor, de dimensões $N_{max}xM$. O tensor *Data* tem o mesmo formato, mas trocando a dimensão N_{max} pelo número de padrões da entrada. Já o tensor da matriz de adjacências, *Neighborhood*, tem o formato do terceiro tensor, de dimensões $N_{max}xN_{max}$. Em cada um, a parte cinza-escuro indica onde estão armazenados os N neurônios existentes atualmente, a parte azul indica onde seriam armazenadas as variáveis associadas a um novo neurônio a ser inserido, e a parte cinza-claro representa espaço de memória não utilizado. 15

3.3 Ilustração do procedimento para retirar um subconjunto das linhas de um tensor, editá-las e reinserí-las em suas posições originais. 17

3.4 Ilustração do procedimento para preservar apenas um subconjunto das linhas de um tensor. 18

4.1 Clustering Error vs Número de dimensões. 21

4.2 Clustering Error vs Tamanho da base de dados. 22

4.3 Clustering Error vs Porcentagem de ruído nos dados. 22

4.4 Tempo médio de execução vs Número de dimensões. 23

4.5 Tempo médio de execução vs Tamanho da base de dados. 23

4.6 Tempo médio de execução vs Porcentagem de ruído nos dados. 24

4.7 Exemplo de base de dados sintética para agrupamento em subespaço com grupos em formato de tubos. Padrões são representados por pontos, e os grupos são indicados pelas diferentes cores. Cada grupo se extende por uma dimensão e se concentra em cada uma das outras. 25
4.8 Clustering Error vs Quantidade de agrupamentos nos dados. 26
4.9 Clustering Error vs Número de dimensões. 27
4.10 Tempo médio de execução vs Quantidade de agrupamentos nos dados. 27
4.11 Tempo médio de execução vs Número de dimensões. 28

Lista de Tabelas

4.1 Intervalos para amostragem de parâmetros do LARFDSSOM, onde P é o tamanho da base de dados. Os valores gerados para a taxa de aprendizagem dos vizinhos foram multiplicados pelas taxas de aprendizagem do vencedor antes de serem usados. 20
4.2 Propriedades das bases de dados do conjunto 4. 25
4.3 Propriedades das bases de dados do conjunto 5. 26

Glossário

N_{max} número máximo de neurônios.

β taxa de aprendizagem das relevâncias.

a_t limite de ativação.

c limite de conexão.

e_b taxa de aprendizagem do vencedor.

e_n taxa de aprendizagem dos vizinhos.

lp porcentagem mínima de vitórias de um grupo.

$maxcomp$ quantidade de competições antes de uma etapa de remoção.

s suavidade do vetor de relevâncias.

t_{max} número de épocas de treinamento.

CE Clustering Error.

CPU Unidade de Processamento Central.

DSSOM Mapa Auto-Organizável Seletivo de Dimensões.

GiB Gibibyte, 2^{30} *bytes* de informação ou armazenamento.

GPU Unidade de Processamento Gráfico.

LARFDSSOM Mapa Auto-Organizável Seletivo de Dimensões com Campo Receptivo Localmente Adaptativo.

LHS Amostragem por Hipercubo Latino.

SOM Mapa Auto-Organizável.

CAPÍTULO 1

Introdução

Neste Capítulo, será feita uma introdução ao trabalho realizado. Serão descritos o contexto e os temas abordados, os objetivos gerais e a estrutura do resto deste documento.

1.1 Contexto e Motivação

Nos últimos anos, a computação vem se tornando cada vez mais presente e importante em várias áreas da sociedade, como comunicação, comércio, educação e entretenimento. Junto com reduções do custo de tecnologias de armazenamento e processamento, esta tendência gera um volume cada vez maior de dados. Há uma demanda crescente por técnicas e algoritmos para processar e extrair conhecimento destes dados, automaticamente e em larga escala, com o objetivo de aprimorar ou criar novas aplicações e serviços para satisfazer as necessidades dos usuários.

A aprendizagem de máquina é a área de pesquisa que contempla boa parte de tais técnicas e algoritmos [1]. Ela geralmente é dividida em dois paradigmas principais: aprendizagem supervisionada e não-supervisionada. A aprendizagem supervisionada tem o objetivo de aproximar funções específicas a partir de conjuntos de dados e respostas esperadas correspondentes, e generalizar essas funções para casos ainda não vistos [1]. Embora seja capaz de obter resultados poderosos, este paradigma tem a desvantagem de precisar das respostas esperadas para todos os dados usados no treinamento. Isto pode ser inviável para conjuntos de dados muito grandes, ou em qualquer situação em que estes rótulos não estejam disponíveis durante o treinamento.

Algoritmos de aprendizagem não-supervisionada extraem informações e características de um conjunto de dados sem nenhuma orientação específica ou conhecimento especializado quanto aos resultados desejados. Eles não tentam reproduzir ou aproximar resultados específicos para cada caso, mas podem utilizar métricas mais abstratas sobre o resultado geral [2]. Tais métricas dependem da tarefa do algoritmo em relação ao conjunto de dados: identificar tendências estatísticas, determinar quais atributos são mais importantes ou descritivos, detectar padrões anômalos ou divergentes do resto do conjunto, etc.

Uma das tarefas de aprendizagem não-supervisionada mais comuns é agrupamento. Dado um conjunto de padrões de dados, deve-se separá-los em grupos coerentes. Há vários critérios diferentes para se definir a coerência de um grupo, como similaridade ou proximidade entre os padrões, distância dos padrões até o centro do grupo, ou densidade da distribuição no espaço de dados.

Na prática, é comum realizar agrupamento em conjuntos de dados com milhares de atributos, em áreas como bioinformática [3], visão computacional [4] ou análise de documentos [5].

A alta dimensionalidade desses conjuntos de dados representa um desafio para algoritmos de agrupamento convencionais, pois alguns atributos irrelevantes dos dados podem dificultar a identificação dos grupos. Além disso, uma alta quantidade de dimensões faz com que os padrões pareçam estar a distâncias muito semelhantes uns dos outros, tornando métricas comuns de distância pouco significativas para o agrupamento [6]. E em algumas aplicações, diferentes grupos podem ser definidos por diferentes conjuntos de dimensões, ou seja, uma dimensão pode ser importante para identificar um grupo, mas não outro. Nesse caso, a tarefa é de agrupamento em subespaço.

Recentemente, há uma grande necessidade de desenvolver implementações eficientes de algoritmos de aprendizagem de máquina e computação científica em geral, visando viabilizar sua utilização em larga escala. Um dos avanços mais importantes é o uso de processamento paralelo de GPUs (Unidades de Processamento Gráfico) no treinamento e utilização de redes neurais artificiais para aprendizagem profunda. Esta técnica tem acelerado em até centenas de vezes o treinamento das redes, que podem chegar a ter milhares de neurônios e milhões de conexões [7]. Teoricamente, dependendo do volume de dados processado, ganhos semelhantes também poderiam ser obtidos com algoritmos de aprendizagem não-supervisionada.

1.2 Objetivos

Este trabalho busca avaliar se programação em GPU pode prover ganhos de desempenho em um algoritmo de agrupamento em subespaço, e determinar sob que condições se pode obter esse ganho. Serão descritas as técnicas e ferramentas utilizadas para reformular o algoritmo, visando paralelizar ao máximo sua execução. Em seguida, a implementação resultante será testada em bases de dados utilizadas para avaliação de algoritmos de agrupamento em subespaço, para avaliar sob que condições se obtém ganhos de desempenho com o uso de GPU.

1.3 Estrutura do Trabalho

No Capítulo 2, serão descritos em detalhes o algoritmo reimplementado neste trabalho, chamado LARFDSSOM, e os algoritmos tomados como base para seu desenvolvimento. No Capítulo 3, será descrita a reimplementação do LARFDSSOM em GPU, bem como os conceitos e ferramentas utilizados. No Capítulo 4, serão descritos os testes realizados e os resultados obtidos ao comparar o desempenho da reimplementação em GPU com uma implementação original do mesmo algoritmo em C++. Finalmente, no Capítulo 5 são apresentadas conclusões sobre o trabalho e possíveis direções futuras.

CAPÍTULO 2

Agrupamento em Subespaços com Mapas Auto-Organizáveis

O Mapa Auto-Organizável (SOM), proposto por Kohonen em [8], é uma rede neural artificial treinada com aprendizagem não-supervisionada que, dada uma base de dados, aprende uma representação discretizada e de baixa dimensionalidade do espaço de entrada. Além de aprender a modelar propriedades topológicas dos dados, esse algoritmo também pode ser utilizado para agrupamento. Porém, ele não funciona para agrupamento em subespaços, quando cada grupo pode considerar um subconjunto diferente das dimensões da entrada e um padrão pode ser atribuído a mais de um grupo.

O Mapa Auto-Organizável Seletivo de Dimensões (DSSOM) foi proposto em [9] para resolver essa limitação ao permitir a cada neurônio atribuir importâncias diferentes a cada dimensão para agrupar os padrões. Esses diferentes níveis de importância são aprendidos ao longo do treinamento, assim como os vetores de pesos. Apesar de ser efetivo para agrupamento em subespaços, seu treinamento é difícil de parametrizar e sua topologia fixa torna difícil adaptar o resultado ao conjunto de dados.

Em [6] foi proposto o Mapa Auto-Organizável Seletivo de Dimensões com Campo Receptivo Localmente Adaptativo (LARFDSSOM), uma extensão de topologia dinâmica do DSSOM na qual neurônios são inseridos e removidos durante o treinamento, e as conexões entre eles são definidas em função da semelhança entre os seus campos receptivos. Em relação ao DSSOM, esse algoritmo possui custo computacional reduzido, parametrização simplificada e melhor qualidade do agrupamento.

Neste Capítulo, serão descritos em detalhe os três algoritmos. Na Seção 2.1 será abordado o Mapa Auto-Organizável original de Kohonen. Na Seção 2.2 o DSSOM e as modificações para adaptá-lo para agrupamento em subespaços. Finalmente, na Seção 2.3 será descrito o LARFDSSOM, suas diferenças em relação aos algoritmos anteriores e sua parametrização.

2.1 Mapa Auto-Organizável

O SOM [8] é uma rede neural artificial cujos neurônios são organizados numa estrutura regular, geralmente uma grade bidimensional de topologia retangular ou hexagonal como na Figura 2.1, e cada neurônio está associado a um vetor de pesos de mesma dimensão dos padrões de entrada. Diferentemente da maioria das redes neurais, o SOM é um algoritmo de aprendizagem competitiva: cada padrão é mapeado para o neurônio cujo vetor de pesos esteja mais próximo em distância euclidiana. Esse mapeamento tenta preservar propriedades topológicas do conjunto de dados, como proximidade entre padrões e distribuição no espaço. Uma forma de pensar nos

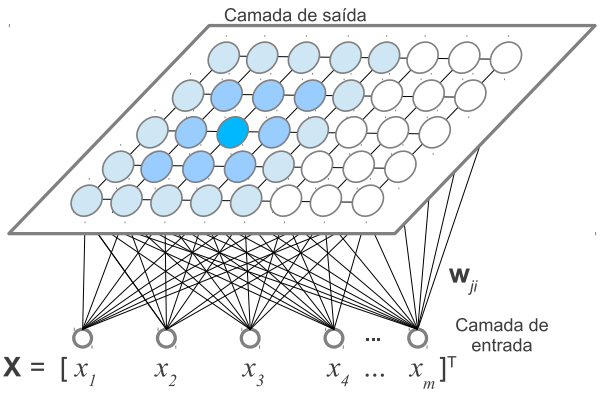

Figura 2.1 Estrutura de um Mapa Auto-Organizável.

vetores de pesos é como sendo protótipos no espaço de entrada que formam uma representação discreta do conjunto de dados. Há mais protótipos em regiões com maior concentração de padrões, e menos em regiões onde a densidade é menor.

A operação do SOM possui duas fases: treinamento e mapeamento. No começo do treinamento cada neurônio i da rede recebe um vetor de pesos W_i. Os vetores de pesos podem ser gerados aleatoriamente ou amostrados do conjunto de entrada. Então, o passo seguinte é repetido inúmeras vezes: um padrão X é escolhido aleatoriamente do conjunto de entrada e apresentado à rede. Para cada neurônio é calculada a distância entre o seu vetor de pesos e o padrão. Então o vetor de pesos do neurônio vencedor i é movido para mais perto do padrão de entrada segundo a Equação 2.1:

$$W_i \leftarrow W_i + e(X - W_i) \tag{2.1}$$

onde e é a taxa de aprendizagem da rede, um valor que define o quanto cada padrão vai se mover ao ser atualizado. Os vetores de pesos dos vizinhos de i na rede também são atualizados da mesma forma, mas em menor grau segundo a Equação 2.2:

$$W_j \leftarrow W_j + Nei(i,j)e(X - W_j) \tag{2.2}$$

onde $Nei(i,j)$ é uma função que diz o quão próximos na rede são os neurônios i e j. Geralmente é uma função gaussiana centrada no neurônio vencedor. A taxa de aprendizagem da rede pode ser uma constante ou pode decrescer ao longo do tempo, para equilibrar exploração e explotação do espaço de busca. A função de vizinhança também pode se estreitar ao longo do tempo.

Durante várias iterações na fase de treinamento, os vetores de pesos da rede convergem para modelar a distribuição dos dados de entrada. Depois, na fase de mapeamento, cada padrão é atribuído ao neurônio com o vetor de pesos mais próximo.

2.2 Mapa Auto-Organizável Seletivo de Dimensões (DSSOM)

Na determinação do neurônio vencedor no SOM, cada dimensão do espaço de entrada tem a mesma importância. Mas no DSSOM, cada neurônio pode atribuir uma relevância diferente

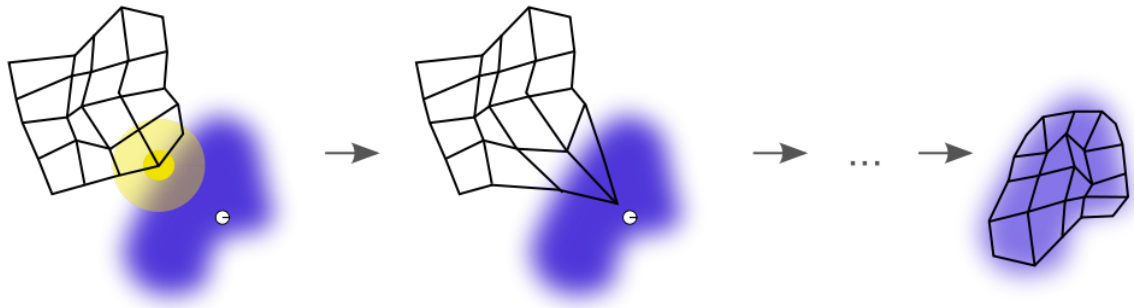

Figura 2.2 Representação do treinamento de um Mapa Auto-Organizável. A massa azul representa o conjunto de padrões de entrada e o ponto branco é o padrão escolhido na iteração atual. Já a grade preta são os vetores de pesos e suas conexões, e o ponto destacado em amarelo é o vetor do vencedor. Ele e seus vizinhos são então movidos para mais próximo do padrão de entrada.
Fonte: Wikipédia (disponível sob licença Creative Commons).

para cada dimensão no cálculo da distância [9]. Especificamente, dado um padrão X, a distância ponderada entre o vetor de pesos de i e o padrão X é dada pela Equação 2.3:

$$D_\omega(X,i) = \sqrt{\sum_{j=1}^{m} \omega_{ij}^2 (X_i - W_{ij})^2} \tag{2.3}$$

onde, ω_{ij} é a relevância dada pelo neurônio i à dimensão j e m é o número de dimensões da entrada. Já a ativação do neurônio é calculada pela Equação 2.4:

$$ac(X,i) = \frac{Sum_i}{D_\omega(X,i) + Sum_i + \varepsilon} \tag{2.4}$$

onde, ε é um valor ínfimo para evitar divisão por zero e Sum_i é a soma dos componentes de ω_i. Durante o treinamento da rede, para cada neurônio i é mantida uma média móvel δ_{ij} da distância na dimensão j entre o vetor de pesos W_i e os padrões agrupados por i. ω_i é calculado em função de δ_i, onde cada valor ω_{ij} é inversamente proporcional à variância na dimensão j dos padrões agrupados pelo neurônio i.

Outra mudança em relação ao SOM original é que o DSSOM é voltado não para redução de dimensionalidade, mas para agrupamento em subespaços. Mais especificamente, cada neurônio é interpretado como um grupo, e um padrão pode ser atribuído a vários grupos, ou a nenhum. Durante as fases de treinamento ou agrupamento final, cada padrão escolhido é apresentado à rede várias vezes, e um vencedor diferente é escolhido toda vez. A atualização dos vetores de peso é semelhante à do SOM, e a ativação dos neurônios a partir do segundo vencedor é calculada de forma diferente. Esses e outros detalhes não foram replicados no LARFDSSOM, então não serão explicados aqui.

2.3 LARFDSSOM

No SOM original e no DSSOM, a quantidade de neurônios é fixa e a estrutura formada por suas conexões também. Já no LARFDSSOM [6], neurônios são inseridos e removidos durante

o treinamento, e as conexões entre eles são atualizadas periodicamente. Assim como no DS-SOM, cada neurônio i está associado a três vetores de mesma dimensão do espaço de entrada: o de pesos (W_i), o de relevâncias (ω_i) e o de distâncias (δ_i), usado somente para calcular o de relevâncias. Adicionalmente, cada neurônio guarda também um contador de vitórias em competições ($wins_i$), usado para escolher quais neurônios serão removidos.

A rede começa com apenas um neurônio, e um padrão da entrada é escolhido aleatoriamente para ser seu vetor de pesos. Seu vetor de distâncias é inicializado com zeros e o de relevâncias com uns. O treinamento do LARFDSSOM tem duas fases: organização e convergência. A fase de organização é ilustrada no Algoritmo 1. Ela consiste de $t_{max} * P$ competições, onde P é o número de padrões de entrada e t_{max} um parâmetro do algoritmo. Em cada competição, assim como nos algoritmos anteriores, um padrão da entrada X é escolhido aleatoriamente e apresentado à rede. A ativação de cada neurônio i é calculada segundo a Equação 2.5:

$$ac(X,i) = \frac{1}{1 + D_\omega(X,i)/(||\omega_i||^2 + \varepsilon)} \tag{2.5}$$

onde, ε é um valor ínfimo para evitar divisão por zero e $||\omega_i||$ é a norma do vetor de relevância. A distância ponderada é segundo a Equação 2.6:

$$D_\omega(X,i) = \sqrt{\sum_{j=1}^{m} \omega_{ij}(X_i - W_{ij})^2} \tag{2.6}$$

O vencedor i e seus vizinhos têm seus vetores de peso atualizados do jeito convencional, segundo a Equação 2.1, onde e é a taxa de aprendizagem apropriada para o neurônio: e_b para o vencedor e e_n para os vizinhos, sendo que o valor usado para e_n é sempre menor que o de e_b. Depois, os vetores de distância são atualizados segundo a Equação 2.7:

$$\delta_i \leftarrow \delta_i + e\beta(|X - W_i| - \delta_i) \tag{2.7}$$

onde, β é um parâmetro para regular a velocidade de mudança da média móvel. Finalmente, são atualizados os vetores de relevância segundo a Equação 2.8:

$$\omega_{ij} \leftarrow \begin{cases} \frac{1}{1+exp(\frac{\delta_{ij} - \delta_{imean}}{s(\delta_{imax} - \delta_{imin})})} & \text{se } \delta_{imin} \neq \delta_{imax} \\ 1 & \text{caso contrário} \end{cases} \tag{2.8}$$

onde, δ_{imin}, δ_{imax} e δ_{imean} são, respectivamente, mínimo, máximo e média entre os componentes do vetor δ_i, e s é um parâmetro para controlar a suavidade do vetor de relevância. É importante observar que, em [6], a fórmula acima está com um erro: δ_{ij} é subtraído de δ_{imean}, ao invés do contrário.

No LARFDSSOM há uma diferença em relação aos outros algoritmos: se a ativação do vencedor da competição não atingir um valor mínimo definido pelo parâmetro a_t, e se o número de neurônios na rede estiver inferior ao máximo definido pelo parâmetro N_{max}, o vencedor e seus vizinhos não são atualizados. Ao invés disso, um novo neurônio é criado, com seu vetor de pesos igual ao padrão de entrada, seu vetor de distâncias preenchido com zeros e o de relevâncias com uns. Ele é então conectado aos neurônios existentes e inserido na rede.

Algoritmo 1 Fase de Auto-Organização

Inicializa os parâmetros do algoritmo: a_t, e_b, e_n, β, s, N_{max}, $maxcomp$, lp, t_{max} e c
Seja P a quantidade de padrões da entrada
$t_{max} \leftarrow t_{max} * P$
$maxcomp \leftarrow maxcomp * P$
O mapa começa com um neurônio i, com W_i inicializado com um padrão qualquer da entrada, δ_i preenchido com zeros, ω_i preenchido com uns, e $wins_i \leftarrow 0$
Seja N a quantidade atual de neurônios na rede
$nwins \leftarrow 1$
para $t = 0$ **até** t_{max} **faça**
  Um padrão X da entrada é escolhido aleatoriamente e apresentado à rede
  Calcula-se a ativação de cada neurônio (Equação 2.5)
  Seja s o id do neurônio vencedor, ou seja, com a maior ativação (a_s)
  se $a_s < a_t$ **e** $N < N_{max}$ **então**
    Um novo neurônio i é criado com $W_i \leftarrow X$, δ_i preenchido com zeros, ω_i preenchido com uns, e $wins_i \leftarrow lp * nwins$
    i é conectado aos neurônios existentes seguindo a condição dada pela Inequação 2.9
  senão
    Os vetores de distâncias δ de s e seus vizinhos são atualizados (Equação 2.7)
    Os vetores de relevâncias ω de s e seus vizinhos são atualizados (Equação 2.8)
    Os vetores de pesos W de s e seus vizinhos são atualizados (Equação 2.1)
    $wins_s \leftarrow wins_s + 1$
  fecha se
  se $nwins = maxcomp$ **então**
    Todos os neurônios i com $wins_i < lp * maxcomp$ são removidos
    Atualiza as conexões de todos neurônios restantes, seguindo a Inequação 2.9
    Reseta os contadores de vitórias dos neurônios restantes: $wins_i \leftarrow 0$
    $nwins \leftarrow 0$
  fecha se
  $nwins \leftarrow nwins + 1$
fecha para

Cada neurônio i mantém um contador $wins_i$ de quantas competições ele ganhou. A cada $maxcomp$ competições, todos os neurônios com menos de $lp * maxcomp$ vitórias são removidos da rede, onde lp e $maxcomp$ são parâmetros do algoritmo para controlar o tamanho mínimo dos grupos e a periodicidade dos *resets*, respectivamente. Observa-se que o parâmetro $maxcomp$ é multiplicado pelo número de padrões da entrada antes de ser utilizado no algoritmo. Depois das remoções, as conexões dos neurônios restantes são atualizadas e seus contadores são resetados para zero. Quando um neurônio é criado, seu contador recebe $lp * wins$ vitórias, onde $wins$ é o número de competições desde o último *reset*.

Quando as conexões de um neurônio são criadas ou atualizadas, ele é conectado a neurônios que dêem importâncias às mesmas dimensões. Especificamente, dois neurônios i e j são conectados se for satisfeita a Inequação 2.9:

$$||\omega_i - \omega_j|| < c\sqrt{m} \tag{2.9}$$

onde, m é o número de dimensões da entrada e c é um parâmetro para regular a conectividade, geralmente setado para $1/2$.

Depois da fase de organização vem a de convergência, que serve para dar um ajuste final aos neurônios restantes. O parâmetro N_{max} é atualizado com a quantidade atual de neurônios na rede (N), e a etapa de remoção é realizada. Em seguida há mais competições, mas sem inserção de novos neurônios. Tudo isso é repetido até que nenhum neurônio seja removido na etapa de remoção, ou até que haja somente um neurônio na rede.

Algoritmo 2 Fase de Convergência

enquanto verdadeiro faça
  $N_{max} \leftarrow N$
  Todos os neurônios i com $wins_i < lp * maxcomp$ são removidos
  se $N = N_{max}$ **ou** $N = 1$ **então**
    retorna
  fecha se
  Atualiza as conexões de todos neurônios restantes, seguindo a Inequação 2.9
  Reseta os contadores de vitórias dos neurônios restantes: $wins_i \leftarrow 0$
  para $t = 0$ **até** t_{max} **faça**
    Um padrão X da entrada é escolhido aleatoriamente e apresentado à rede
    Calcula-se a ativação de cada neurônio (Equação 2.5)
    Seja s o id do neurônio vencedor
    Os vetores de distâncias δ de s e seus vizinhos são atualizados (Equação 2.7)
    Os vetores de relevâncias ω de s e seus vizinhos são atualizados (Equação 2.8)
    Os vetores de pesos W de s e seus vizinhos são atualizados (Equação 2.1)
    $wins_s \leftarrow wins_s + 1$
  fecha para
fecha enquanto

Finalmente, para definir quais padrões pertencem a quais grupos, é realizado o agrupamento descrito pelo Algoritmo 3. Para cada padrão é calculada a ativação dos neurônios, e ele é

atribuído a todos cuja ativação atinja o limiar a_t. Se o algoritmo estiver configurado para realizar agrupamento projetivo, cada padrão é atribuído a no máximo um grupo. Além disso, o mecanismo de filtragem de ruído pode ser desativado, fazendo que cada padrão seja atribuído a pelo menos um grupo.

Algoritmo 3 Agrupamento Final

para cada padrão X da entrada **faça**
  X é apresentado à rede
  Calcula-se a ativação de cada neurônio (Equação 2.5)
  Seja s o id do neurônio vencedor, ou seja, com a maior ativação (a_s)
  se $a_s \geq a_t$ **então**
    repete
      Atribui X ao agrupamento representado pelo neurônio vencedor s
      se o algoritmo estiver realizando agrupamento projetivo **então**
        Sai do laço
      fecha se
      Encontra-se o próximo neurônio s com maior ativação a_s entre os restantes
    até que $a_s < a_t$
  senão
    se o algoritmo estiver realizando filtragem de ruído **então**
      Atribui X ao conjunto dos *outliers*
    senão
      Atribui X ao agrupamento representado pelo neurônio vencedor s
    fecha se
  fecha se
fecha para

Neste Capítulo, foram descritos o treinamento e utilização do LARFDSSOM, bem como os algoritmos com base nos quais ele foi desenvolvido. No próximo Capítulo, será explicado como o LARFDSSOM foi reimplementado para utilizar processamento paralelo em GPU.

CAPÍTULO 3

Implementação do LARFDSSOM em GPU

Na primeira Seção deste Capítulo, serão explicados os fundamentos de programação para GPU, bem como as condições necessárias para se obter ganhos de desempenho. Na Seção 3.2, serão descritos a linguagem de programação e o *framework* utilizados para a implementação do algoritmo. Na Seção 3.3, será apresentado um exemplo prático de uso do Torch. Finalmente, na Seção 3.4 será descrita a modelagem do algoritmo usando o *framework* escolhido, bem como os detalhes de sua implementação.

3.1 Programação com GPUs

Unidades de Processamento Gráfico (GPUs) são unidades de *hardware* dedicadas originalmente para tarefas de computação gráfica, como renderização, mas que são cada vez mais utilizadas para computação de propósito geral. Diferentemente das CPUs, as unidades de processamento convencionais dos computadores, as GPUs possuem núcleos mais simples e menos potentes, mas em quantidade muito maior, tipicamente centenas ou milhares. Estes núcleos executam paralelamente as mesmas operações em uma quantidade massiva de dados, o que em CPU tipicamente seria realizado iterativamente em um ou mais laços. Isto faz com que as GPUs sejam muito eficientes para computação paralelizável, principalmente operações com vetores e matrizes muito grandes.

A transferência de dados da memória RAM para a GPU e vice-versa representa um *overhead* não-trivial na execução de um programa, então há algumas condições para se obter ganhos reais de desempenho. É necessário reformular o algoritmo para agrupar as operações em lotes o máximo possível. Os dados a serem manipulados devem estar em grande quantidade, com poucas dependências entre eles na execução. Finalmente, é necessário que haja muitas operações a serem realizadas, com alto grau de paralelismo e fluxo sequencial, com poucos ou nenhum laços ou operações condicionais.

3.2 Linguagem de programação e *Framework*

A implementação do LARFDSSOM em GPU foi desenvolvida usando Torch [10], um *framework* de código aberto para computação científica. O Torch é amplamente utilizado em aprendizagem de máquina, especialmente em redes neurais artificiais para aprendizagem profunda [7], e possui um vasto ecossistema de bibliotecas para vários algoritmos e aplicações.

Torch é feito em Lua, uma linguagem de programação de propósito geral com tipagem dinâmica e interface para executar diretamente código em C, e é executado com LuaJIT, um

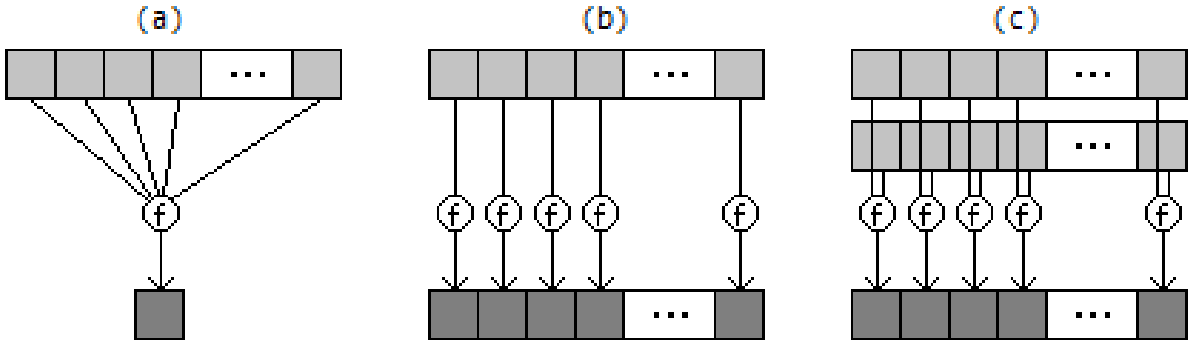

Figura 3.1 Operações comuns com tensores no *framework* Torch: (a) *Reduce*: calcular uma função que agrega todos os elementos do tensor, como soma, máximo ou mínimo; (b) *Map*: aplicar uma função independentemente para cada elemento, como multiplicação por um escalar ou elevar ao quadrado; (c) *Zip* e *Map*: calcular uma função elemento-a-elemento entre dois tensores, como soma ou subtração.

compilador *Just-In-Time* extremamente rápido. Todos os componentes fundamentais do Torch possuem suporte para uso de GPUs através de CUDA, uma tecnologia proprietária para GPUs da NVIDIA que serve como interface para controlar os dispositivos. O *framework* é robusto e livra o usuário de ter que lidar com as primitivas de baixo nível dessa interface.

O bloco fundamental do Torch é o tensor, uma flexível e eficiente estrutura de dados que representa um *array* numérico multidimensional. Um tensor generaliza os conceitos de vetores e matrizes e possui uma quantidade arbitrária de dimensões, cada uma com tamanhos arbitrários. Esse *array* pode ser de inteiros de um, dois ou quatro *bytes*, ou de números de ponto flutuante de precisão simples ou dupla. Torch implementa operações simples e eficientes para indexar, redimensionar, copiar e fatiar tensores, bem como operações matemáticas fundamentais para vetores e matrizes. Além disso, para executar operações em GPU, pode-se utilizar tensores cujos dados são armazenados em memória de GPU. É simples transferir o conteúdo de um tensor da memória RAM do computador para a GPU, e vice-versa.

3.3 Exemplo de uso de tensores

O tensor é uma abstração matemática muito útil e intuitiva para várias aplicações, como álgebra linear, otimização numérica e redes neurais artificiais. A maioria das operações matemáticas importantes implementadas em Torch são de uma das formas mostradas na Figura 3.1. Normalmente essas operações seriam realizadas em um laço, mas com uma quantidade suficiente de núcleos e espaço em memória, elas podem ser facilmente paralelizadas. As operações de *Zip* e *Map* podem ser realizadas em um só passo, enquanto o *Reduce* pode ser completado em menos passos do que o normal.

O seguinte trecho de código em Lua calcula a distância ponderada entre um padrão de entrada x e os neurônios da rede:

```
1  local weighted_dists = {}
2  for i = 1, N do
3          local norm = 0
```

```
4           for j = 1, M do
5                   local dif = (x[j] - weights[i][j])^2
6                   norm = norm + relevances[i][j]*dif
7           end
8           weighted_dists[i] = math.sqrt(norm)
9   end
```

Nesse código, N e M são as quantidades de neurônios na rede e dimensões da entrada, respectivamente. x é uma lista de valores numéricos de cada atributo ou dimensão do padrão. *weights* e *relevances* são listas que guardam, pra cada neurônio, os vetores de pesos e relevâncias, que por sua vez são listas de valores, assim como x. *norm* e dif são variáveis numéricas para armazenar valores temporários. *weighted_dists* é a lista das distâncias ponderadas finais para cada neurônio. O código acima é sequencial e realiza dois laços, o externo iterando sobre os neurônios e o interno sobre as dimensões.

O seguinte trecho de código tem exatamente a mesma função do anterior, mas utiliza tensores e operações do Torch:

```
1  local repeated_x = x
2          :view(1, M)
3          :expand(N, M)
4  local dif = repeated_x - weights
5  dif:pow(2)
6  dif:cmul(relevances)
7  local weighted_dists = torch.sum(dif, 2)
8          :squeeze(2)
9  weighted_dists:sqrt()
```

Aqui, N e M mantêm o mesmo significado, e todas outras variáveis são tensores numéricos com diferentes tamanhos e dimensões. x é um vetor de tamanho M e *repeated_x* é uma matriz NxM, onde o vetor x é repetido em cada linha. *weights* e *relevances* são matrizes NxM, onde cada linha i guarda um vetor de pesos ou relevâncias associado ao neurônio i. dif é uma matriz NxM que guarda as diferenças, pra cada dimensão, entre cada neurônio e o padrão de entrada. *weighted_dists* é um vetor de tamanho N que guarda o resultado final pra cada neurônio.

As operações nas linhas 2 e 3 repetem o vetor x numa matriz para compará-lo com o vetor de pesos de cada neurônio. Essas operações são bastante eficientes, pois não realizam replicação real de memória. A operação da linha 4 subtrai dois tensores, elemento a elemento, numa operação de Zip e Map, calculando a diferença entre o padrão x e o vetor de pesos de cada neurônio. Na linha 5, cada elemento do tensor é elevado ao quadrado, numa operação de Map. Em seguida, eles são multiplicados pelos vetores de relevâncias de cada neurônio, elemento a elemento, em outra operação de Zip e Map.

Na linha 7, é calculada a soma dos elementos de cada linha do tensor dif. Ou seja, uma operação de *Reduce* é executada para cada linha de um tensor bidimensional NxM, e o resultado é um tensor unidimensional de tamanho N. Finalmente, na linha 9 é calculada a raiz quadrada de cada soma do vetor, obtendo o resultado final para cada neurônio.

Nota-se que em todas as operações entre dois tensores diferentes, eles possuem os mesmos tamanhos e dimensões. Nesse código não há laços explícitos: eles estão implícitos em cada operação realizada entre tensores. Quando os conteúdos dos tensores são armazenados em GPU, o *framework* "desenrola" os laços e paraleliza as operações o quanto for possível,

dependendo do volume de dados e da quantidade de núcleos disponíveis na GPU.

A princípio, o código em Torch não ficou menor ou mais simples que o original, mas ficou mais matematicamente intuitivo e semelhante às descrições formais do algoritmo. Ao abstrair os laços e índices, o usuário pode focar nas operações realizadas e cometer menos erros relacionados à matemática.

3.4 Implementação do LARFDSSOM usando tensores

A tarefa mais importante ao implementar um algoritmo utilizando Torch é modelar seus dados como tensores, não só para facilitar o paralelismo na execução mas também para deixar a implementação mais intuitiva e matematicamente conveniente. Nesta Seção serão descritas a modelagem das variáveis do algoritmo e a reformulação de seus subprocedimentos mais importantes.

3.4.1 Variáveis

Sejam P a quantidade de padrões da entrada, N a quantidade atual de neurônios na rede e M a quantidade de dimensões do espaço de entrada. A entrada do algoritmo é uma lista de vetores de tamanho M. Cada neurônio possui um contador de vitórias, uma lista de no máximo N conexões e três vetores de dimensão M. Essas são as principais variáveis usadas no algoritmo.

Os contadores de vitórias são utilizados em quatro circunstâncias:

- Inserção: o valor do novo neurônio é definido
- Competição: o valor do vencedor é incrementado
- Remoção: são selecionados aqueles neurônios com valor maior ou igual a $lp * maxcomp$
- *Reset* de vitórias: todos os valores são zerados

As duas primeiras operações mexem com apenas um valor cada, então são triviais. Já as outras duas são perfeitamente paralelizáveis, então faz sentido armazenar todos os contadores em um tensor.

Em um grafo, há diversas formas de se armazenar as conexões existentes, todas com suas vantagens e desvantagens: matrizes de incidência, matrizes de adjacência, listas de incidência, listas de adjacência, etc. No algoritmo, as conexões dos neurônios são utilizadas em três circunstâncias:

- Quando um neurônio é criado, ele é conectado a alguns dos existentes
- Em uma atualização, são selecionados os vizinhos do vencedor
- Em uma remoção, são recalculadas todas as conexões de todos os neurônios restantes

Para definir se dois neurônios serão conectados, compara-se os seus vetores de relevância. Essa operação pode ser paralelizada nos primeiro e terceiro casos, dando como resultado o valor

da semelhança obtido em cada comparação. No primeiro caso tem-se vetor unidimensional, e no terceiro, uma matriz. Os tensores do Torch implementam funções de comparação que dão como resultado tensores de booleanos. Matrizes de adjacência são armazenadas como matrizes de booleanos, então essa foi a forma escolhida para se armazenar as conexões dos neurônios. Além de parecer mais natural, é uma forma mais adequada para operações em paralelo, inclusive a do segundo caso.

Ainda restam os vetores dos padrões de entrada e os três vetores de cada neurônio: o de pesos, o de distâncias e o de relevâncias. Em geral, as operações com eles são operações vetoriais comuns, então também faz sentido armazenar esses vetores como tensores, para se paralelizar as operações. Além disso, muitas operações no algoritmo são repetidas para vários neurônios, portanto faz sentido agrupar em um mesmo tensor o mesmo vetor de cada neurônio.

Finalmente, os seguintes tensores foram utilizados para modelar as variáveis do algoritmo:

- *Wins*: um tensor de dimensão *N* de inteiros, com os contadores de vitórias
- *Neighborhood*: um tensor *NxN* de booleanos, representando as conexões existentes
- *Weights*: um tensor *NxM* de números reais, com os vetores de pesos
- *Distances*: um tensor *NxM* de números reais, com os vetores de médias de distância
- *Relevances*: um tensor *NxM* de números reais, com os vetores de relevâncias
- *Relevance_norms*: um tensor de dimensão *N* de números reais, com a norma de cada vetor de relevância, recalculado adequadamente a cada atualização
- *Data*: um tensor *PxM* de números reais, com os padrões da entrada

É importante observar que a quantidade de neurônios da rede muda ao longo do treinamento. Para lidar com isso, para cada tensor foi alocado o tamanho para a quantidade máxima, dada pelo parâmetro N_{max}. Os valores existentes eram armazenados nos primeiros índices, para que ficasse fácil cortar a "fatia" útil do tensor. A Subseção seguinte explica como se lidou com inserções e remoções.

3.4.2 Treinamento

A fase de treinamento do algoritmo é, basicamente, uma sequência de competições por padrões de entrada e remoções de neurônios. A cada competição, um neurônio pode ser adicionado à rede ou um ou mais neurônios podem ser atualizados, dependendo da ativação do vencedor. Nas remoções, a escolha dos neurônios removidos depende de quantas competições cada um ganhou. O resultado de uma competição depende das competições e remoções anteriores, e o resultado das remoções também. Portanto, elas são inevitavelmente seriais e não há como paralelizá-las. Com isso, é possível apenas otimizar as operações internas desses subprocedimentos, que serão abordados um a um.

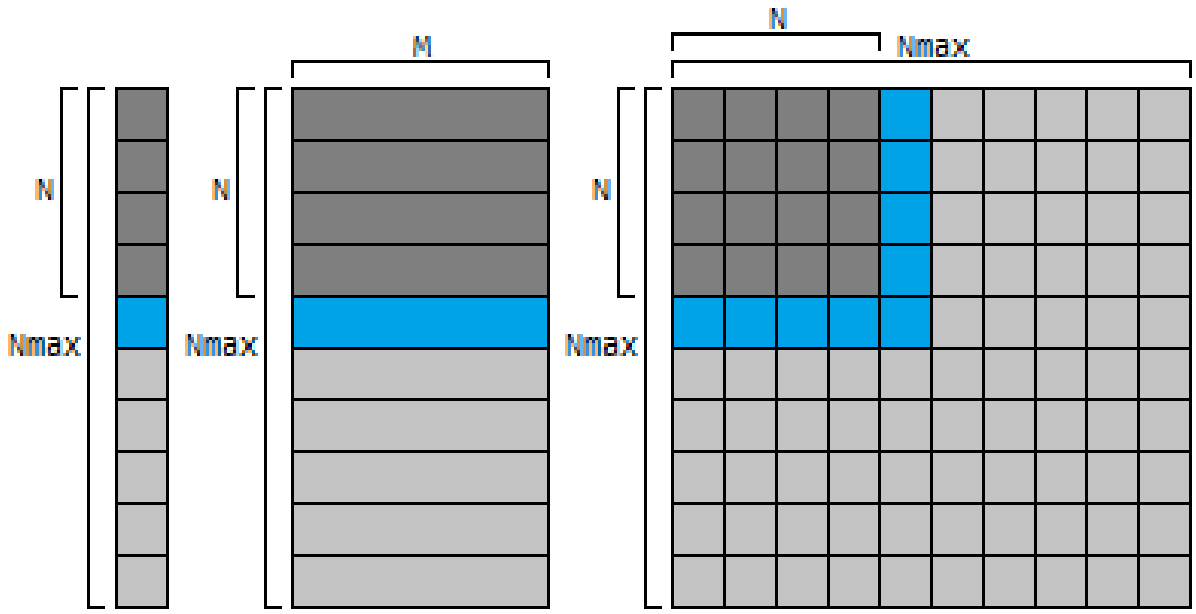

Figura 3.2 Ilustração dos tensores usados para armazenar as variáveis associadas aos neurônios. Os tensores dos contadores de vitórias (*Wins*) e das normas dos vetores de relevância (*Relevance_norms*) têm o formato do tensor unidimensional à esquerda. Os tensores *Weights*, *Distances* e *Relevances* têm o formato do segundo tensor, de dimensões $N_{max}xM$. O tensor *Data* tem o mesmo formato, mas trocando a dimensão N_{max} pelo número de padrões da entrada. Já o tensor da matriz de adjacências, *Neighborhood*, tem o formato do terceiro tensor, de dimensões $N_{max}xN_{max}$. Em cada um, a parte cinza-escuro indica onde estão armazenados os N neurônios existentes atualmente, a parte azul indica onde seriam armazenadas as variáveis associadas a um novo neurônio a ser inserido, e a parte cinza-claro representa espaço de memória não utilizado.

3.4.2.1 Competição

Primeiro, se escolhe aleatoriamente um padrão da entrada, e então é calculada a ativação de cada neurônio, usando seus vetores de pesos e relevâncias. As operações internas desse procedimento são operações vetoriais comuns e a ativação de cada neurônio é independente, podendo então ser paralelizado. No final, o resultado é um tensor de valores de ativação, do qual é preciso somente selecionar o maior valor, operação também paralelizável. Se esse valor for igual ou superior ao parâmetro a_t, o algoritmo irá para o procedimento de atualização. Senão, e se a quantidade de neurônios não estiver no máximo indicado pelo parâmetro N_{max}, será realizada a inserção de um novo neurônio.

3.4.2.2 Inserção de Neurônios

Como foi dito anteriormente, os tensores que armazenam as variáveis dos neurônios são alocados com tamanho máximo, e os valores existentes são armazenados nos primeiros índices. Portanto, para adicionar novos elementos, é preciso apenas colocá-los nas próximas posições em cada tensor, como na Figura 3.2. O contador de vitórias do novo neurônio será setado com lp vezes o número de vitórias desde o último *reset*, o vetor de pesos será copiado do padrão atual da competição, o de distâncias médias será preenchido com zeros e o de relevâncias com uns. Finalmente, para definir as conexões, é preciso calcular a semelhança entre o vetor de relevâncias do novo neurônio e o de cada neurônio existente e comparar cada valor resultante com o valor $c\sqrt{m}$, o que também pode ser feito em paralelo.

3.4.2.3 Atualização do Mapa

Depois de cada competição, o neurônio vencedor tem seu contador de vitórias incrementado, e ele e seus vizinhos são atualizados. Seus vetores de peso são movidos para mais perto do padrão de entrada, e seus vetores de distância e relevância são atualizados para considerar também o padrão de entrada em seus campos receptivos.

Incrementar o contador de vitórias do vencedor é uma operação trivial. Já as outras operações são um pouco mais complicadas, porque são realizadas em apenas um subconjunto dos neurônios, a vizinhança do vencedor. Felizmente, os tensores do Torch contam com uma operação eficiente para extrair um subconjunto das linhas de uma matriz, dada uma lista de índices. No caso, essa lista indica a vizinhança do vencedor, obtida das posições positivas da linha do vencedor na matriz de conexões. Depois de alterados, os vetores podem ser reinseridos em suas respectivas posições no tensor original com outra operação que usa a mesma lista de índices, como na Figura 3.3.

As operações para atualizar os vetores de pesos e de distâncias dos neurônios da vizinhança são triviais de paralelizar. É necessário apenas preparar um tensor com as taxas de aprendizagem, preenchido com e_n em todas as posições, exceto a do vencedor, com e_b. Por outro lado, com os vetores de relevâncias, não é tão simples, pois há um teste de condição: se todos os elementos do vetor de distâncias forem iguais, o vetor de relevâncias é preenchido com uns, caso contrário é aplicada a segunda parte da Equação 2.8. Para resolver esse problema, é preciso reutilizar as operações mencionadas anteriormente para, dada uma lista de índices, selecionar um subconjunto do tensor. A ideia é calcular e comparar o máximo e mínimo de cada vetor de

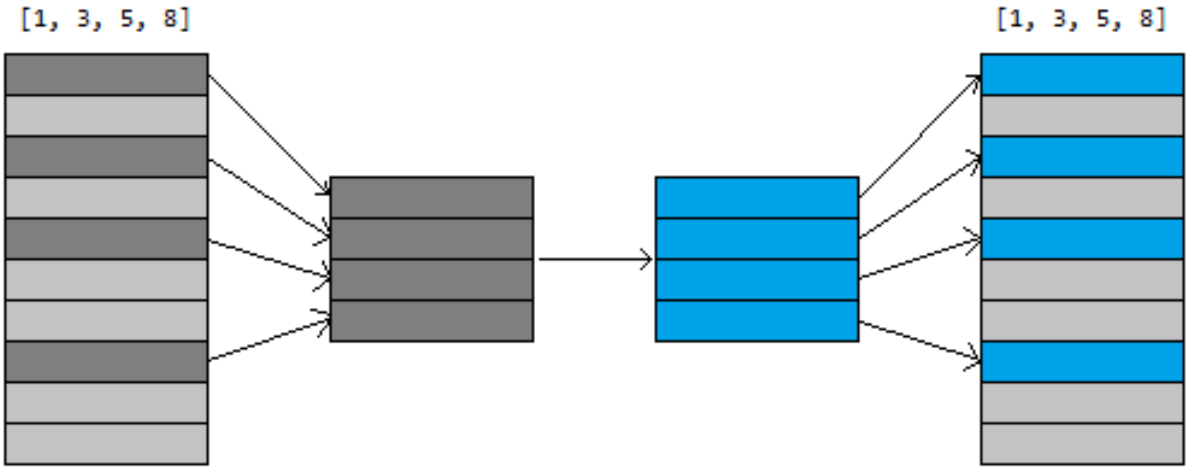

Figura 3.3 Ilustração do procedimento para retirar um subconjunto das linhas de um tensor, editá-las e reinserí-las em suas posições originais.

distâncias, separar do tensor as linhas que não serão preenchidas com uns, processá-las separadamente e reinserí-las. Os outros vetores podem ser preenchidos com uns já em seus devidos lugares.

3.4.2.4 Remoção de Neurônios

Dado o tensor de contadores de vitórias, pode-se comparar os elementos um a um com o valor $lp * maxcomp$ para se obter os índices dos neurônios que serão preservados na rede. Depois, para cada um dos outros tensores, exceto o da entrada e o das conexões, é necessário apenas separar os subconjuntos dos neurônios restantes e reinserí-los nos primeiros índices de seus respectivos tensores, como na Figura 3.4. Depois é preciso recalcular as conexões entre cada par de neurônios, o que é muito semelhante ao que é feito na inserção, e portanto também sendo realizado em paralelo.

3.4.3 Agrupamento Final

Finalmente, são calculadas a ativação de cada neurônio para cada padrão de entrada, o que também pode ser feito em paralelo. Então, para cada padrão é preciso selecionar os índices dos neurônios com ativação igual ou superior a a_t, e isso também em paralelo. Porém, se o algoritmo estiver configurado para realizar agrupamento projetivo, cada padrão poderá ser agrupado apenas pelo neurônio de maior ativação. Ou se o filtro de ruído estiver desativado, cada padrão terá que receber pelo menos um grupo, incluindo aqueles para os quais nenhuma ativação atingiu a_t. Em ambos os casos, é preciso procurar o neurônio com a maior ativação para cada padrão, o que também é paralelizável.

Outra técnica importante para garantir o desempenho da implementação foi a reutilização de tensores. Muitas das operações com tensores precisavam alocar memória na GPU para armazenar resultados intermediários, o que é custoso. Então para cada operação realizada, fora

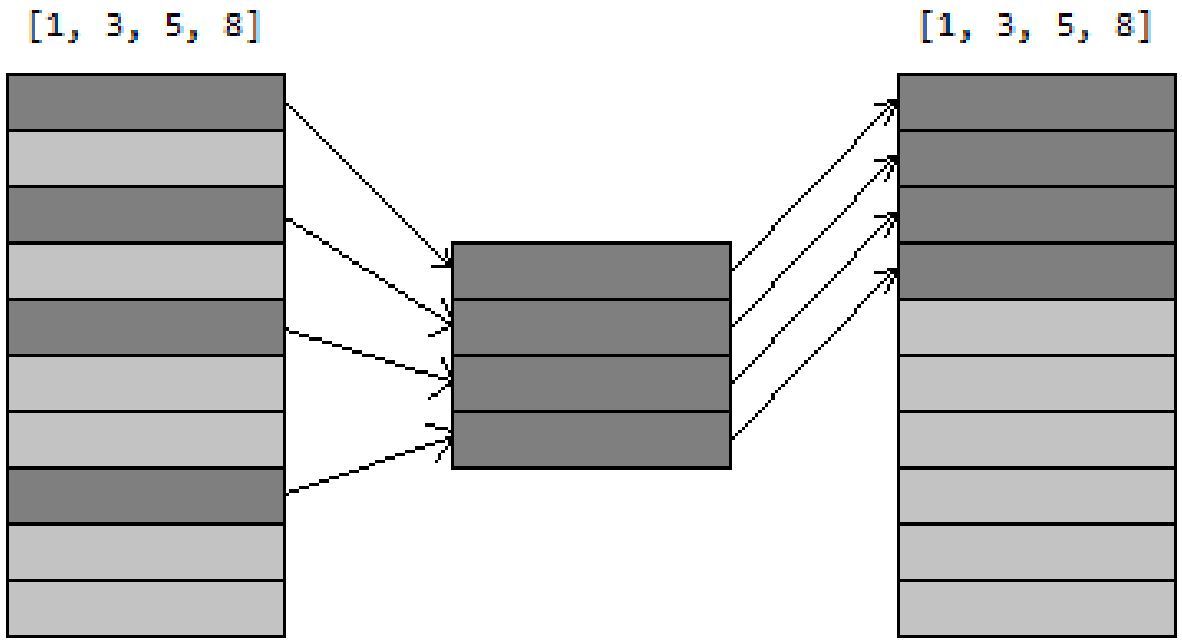

Figura 3.4 Ilustração do procedimento para preservar apenas um subconjunto das linhas de um tensor.

criado um tensor pré-alocado com tamanho suficiente para guardar o resultado. Assim, todas as alocações de memória para tensores acontecem já no começo do algoritmo.

Neste Capítulo, foi descrito como o algoritmo LARFDSSOM foi reimplementado utilizando o *framework* Torch para utilizar processamento paralelo em GPU. No próximo Capítulo, serão descritos os testes realizados para avaliar o ganho de desempenho obtido em diferentes condições.

CAPÍTULO 4

Testes

Neste Capítulo, serão descritos os testes realizados para avaliar o ganho de desempenho da implementação do LARFDSSOM em GPU, bem como os resultados obtidos. Na Seção 4.1, serão descritas as bases de dados utilizadas, a parametrização do algoritmo, a metodologia experimental e os ambientes dos testes. Na Seção 4.2, serão avaliados os resultados obtidos ao se comparar as implementações com os conjuntos de dados utilizados em [6]. Finalmente, na Seção 4.3, serão avaliados os resultados com os conjuntos de dados sintéticos gerados para este trabalho.

4.1 Configurações dos Testes

Para comparar o desempenho da implementação em Torch com o da implementação original, em C++, foram utilizadas bases de dados sintéticas separadas em conjuntos, cada um dedicado a avaliar a diferença de desempenho em função de uma característica das bases. As bases de dados dos conjuntos de 1 a 3 foram as mesmas usadas em [6] para avaliar o LARFDSSOM. Essas bases são disponibilizadas no OpenSubspace [2], um *framework* para avaliação de algoritmos de agrupamento em subespaços, e variam em quantidade de padrões (conjunto 1), quantidade de dimensões (conjunto 2) e porcentagem de ruído nos dados (conjunto 3). Além disso foram criadas bases de dados artificiais, variando a quantidade de grupos nos dados (conjunto 4) e a quantidade de dimensões, em escala maior (conjunto 5).

Os testes foram realizados não só para comparar o tempo de execução das implementações, mas também para conferir se a implementação em GPU produz resultados com a mesma qualidade que a implementação original. A métrica utilizada para avaliar isso foi o Clustering Error (CE) [2], utilizado também em [6] para avaliar a precisão dos resultados do LARFDSSOM. Dados dois conjuntos de grupos em subespaços, um original e verdadeiro e outro encontrado por um algoritmo, procura-se um emparelhamento entre os grupos de cada conjunto que maximize a semelhança total entre cada par de grupos. A semelhança entre dois grupos é definida pelo conjunto de padrões pertencentes a cada um e ao conjunto de dimensões do espaço de dados que são relevantes para cada um. A qualidade total do emparelhamento define o valor do CE, cujo valor varia entre 0 e 1, onde 1 é um emparelhamento perfeito.

Primeiro foi preciso definir, para cada base de dados, os parâmetros do LARFDSSOM usados na execução. Isso depende muito das características de cada base, como tamanho ou número de grupos. Foram gerados aleatoriamente 500 conjuntos de parâmetros amostrados usando Amostragem por Hipercubo Latino (LHS) [11], uma técnica de amostragem inspirada em análise de sensibilidade que tenta cobrir o máximo possível o espaço de parâmetros. Dado

Tabela 4.1 Intervalos para amostragem de parâmetros do LARFDSSOM, onde P é o tamanho da base de dados. Os valores gerados para a taxa de aprendizagem dos vizinhos foram multiplicados pelas taxas de aprendizagem do vencedor antes de serem usados.

Parâmetro	Mínimo	Máximo
limite de ativação (a_t)	0.9	0.999
porcentagem mínima de vitórias de um grupo (lp)	0.01%	10%
taxa de aprendizagem das relevâncias (β)	0.0001	0.5
quantidade de competições antes de uma etapa de remoção ($maxcomp$)	1*P	100*P
taxa de aprendizagem do vencedor (e_b)	0.0001	0.01
taxa de aprendizagem dos vizinhos (e_n)	0.001	0.1
suavidade do vetor de relevâncias (s)	0.01	0.05
limite de conexão (c)	0	0.5

um intervalo de amostragem para cada parâmetro, essa técnica divide cada intervalo em N partes iguais, onde N é o número de amostras desejadas, extrai uma amostra de cada intervalo e combina aleatoriamente as amostras entre diferentes parâmetros, formando N conjuntos. Os intervalos de amostragem de cada parâmetro do LARFDSSOM estão na Tabela 4.1.

Depois de gerados os conjuntos de parâmetros, a implementação original foi executada com cada um para cada base de dados, e para cada base foi escolhido o conjunto que obteve o melhor valor de CE. Porém, para as bases de dados do conjunto 5 foi escolhido o mesmo conjunto de parâmetros, aquele que obteve o melhor CE com a menor base, com 32 dimensões. Para todas as bases, o número de épocas de treinamento (t_{max}) ficou fixo em 100 vezes o número de padrões da base. Já o número máximo de neurônios (N_{max}) ficou fixo em 70 para as bases de dados dos conjuntos de 1 a 3, assim como em [6], e igual à quantidade verdadeira de grupos para as bases dos outros conjuntos. Cada implementação foi executada com cada base 30 vezes, variando a semente do gerador aleatório, e foram coletadas médias e desvios padrão do tempo de execução e do CE obtido.

Os testes com as bases de dados do conjunto 5 foram realizados numa máquina virtual p2.xlarge da plataforma Amazon Web Services, com sistema operacional Amazon Linux, processador Intel Xeon E5-2686 v4 com 4 núcleos e 2.30GHz, 61GiB de memória RAM e GPU NVIDIA Tesla K80 com 2496 núcleos e 12GiB de memória. Já os testes com as bases dos outros conjuntos foram realizados numa máquina virtual NC6 da plataforma Microsoft Azure, com sistema operacional Ubuntu 16.04, processador Intel Xeon E5-2690 v3 com 6 núcleos e 2.60GHz, 56GiB de memória RAM e mesma GPU.

4.2 Testes com Bases de Dados do OpenSubspace

Nesta Seção, são comparados os resultados obtidos pelas duas implementações nas bases de dados do OpenSubspace, que variam em quantidade de dimensões, tamanho da base de dados e porcentagem de ruído nos dados. Na Subseção 4.2.1 é avaliada a qualidade dos agrupamentos encontrados, e na Subseção 4.2.2 é avaliado o tempo de execução.

4.2.1 Qualidade do Agrupamento

As Figuras 4.1, 4.2 e 4.3 comparam os valores médios de CE obtidos. Percebe-se que a implementação em GPU obtém resultados semelhantes aos da implementação original em quase todos os casos, exceto na base com 70% de ruído, onde o valor obtido foi inferior em 0,08. O desvio padrão obtido pelas implementações original e em GPU foram de 0,04 e 0,06 respectivamente, então esse erro foi considerado dentro do esperado, considerando as diferenças acumuladas ao longo das iterações devido a imprecisões na representação de ponto flutuante, e também a diferenças na seleção aleatória dos padrões da entrada para as competições.

É importante observar que esses resultados foram inferiores aos apresentados em [6]. Isso aconteceu porque em [6], para cada base de dados, mostra-se o melhor CE obtido entre 500 conjuntos de parâmetros diferentes, ao executar o LARFDSSOM apenas uma vez. Dessa forma, os melhores resultados são naturalmente afetados pela aleatoriedade do algoritmo. Já aqui, depois de selecionado o melhor conjunto de parâmetros para cada base, foram coletadas as médias do CE obtido em 30 execuções, dando resultados mais robustos.

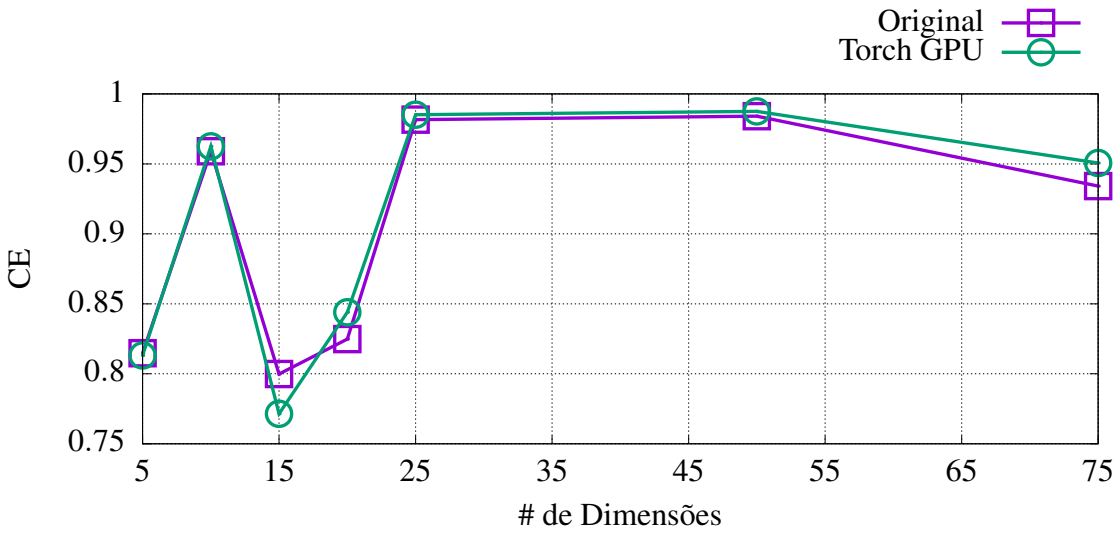

Figura 4.1 Clustering Error vs Número de dimensões.

4.2.2 Tempo de Execução

As Figuras 4.4, 4.5 e 4.6 comparam os tempos médios de execução. Para todas as bases de dados a implementação em GPU foi notavelmente pior, em cerca de duas ordens de grandeza. Não foi identificada nenhuma tendência em especial ao se observar o desempenho em função do nível de ruído.

Na Figura 4.4, nas bases com até 25 dimensões não foi possível perceber nenhuma correlação significativa com o tempo de execução. Provavelmente esse número de dimensões ainda é muito baixo para afetar significativamente o tempo de execução do algoritmo, sendo ofuscado pela aleatoriedade na geração das bases de dados.

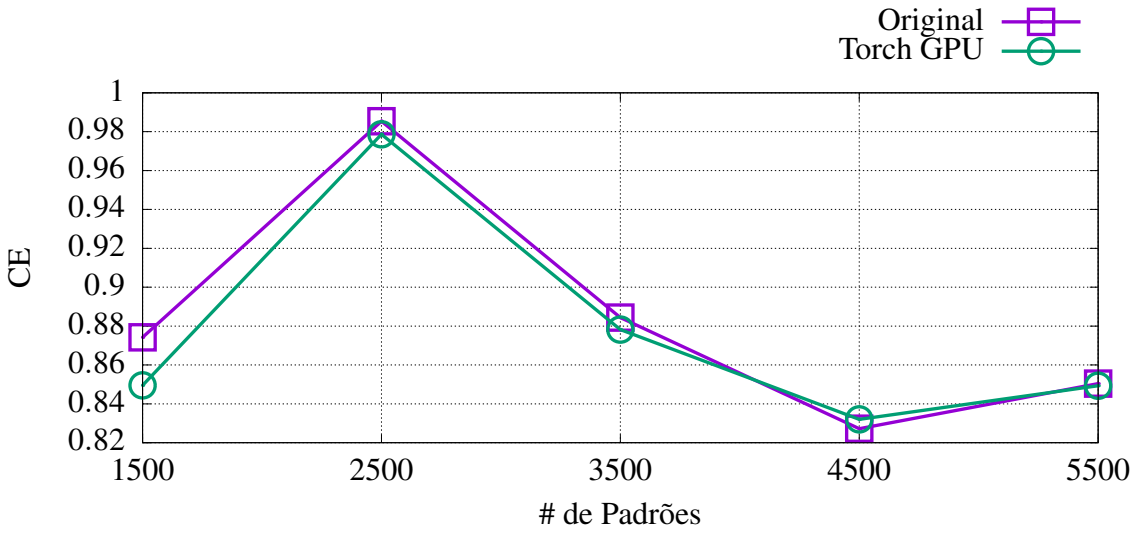

Figura 4.2 Clustering Error vs Tamanho da base de dados.

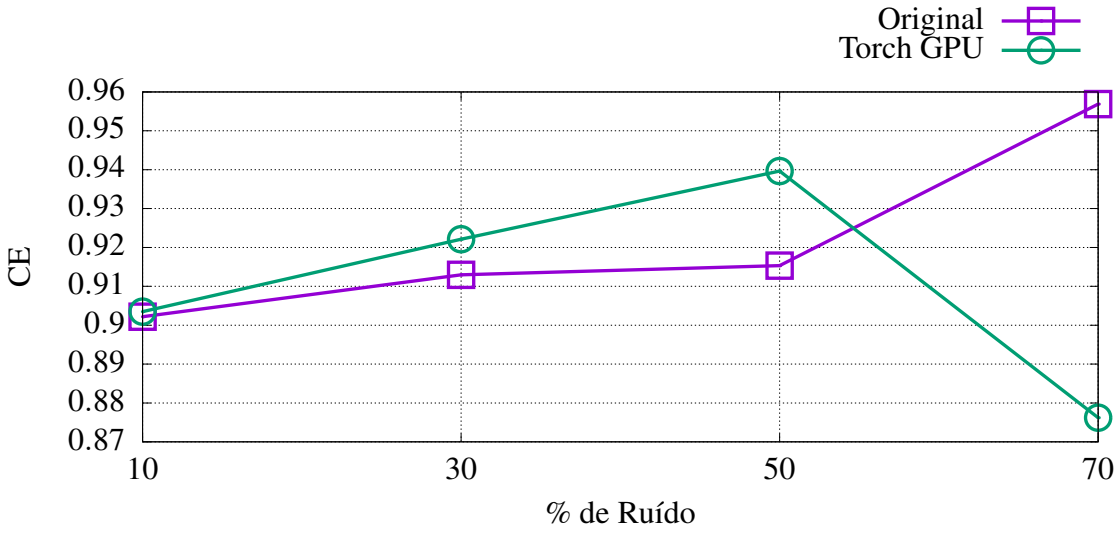

Figura 4.3 Clustering Error vs Porcentagem de ruído nos dados.

A partir de 25 dimensões, o tempo de execução da implementação original apresenta uma tendencia de aumento, enquanto o da implementação em GPU não mostra crescimento perceptível. Como esperado, o paralelismo da GPU deve estar mitigando o efeito da dimensionalidade no tempo de execução. Infelizmente, as bases de dados mostradas aqui não possuem dimensões o bastante para se obter vantagens com a implementação em GPU. Na Seção 4.3, serão mostrados testes com bases de dados com até milhares de dimensões, onde as diferenças se mostram de forma mais clara.

Na Figura 4.5, os tempos de execução apresentam tendência de crescimento em função do número de padrões nas bases de dados. O tempo da implementação com GPU não parece cres-

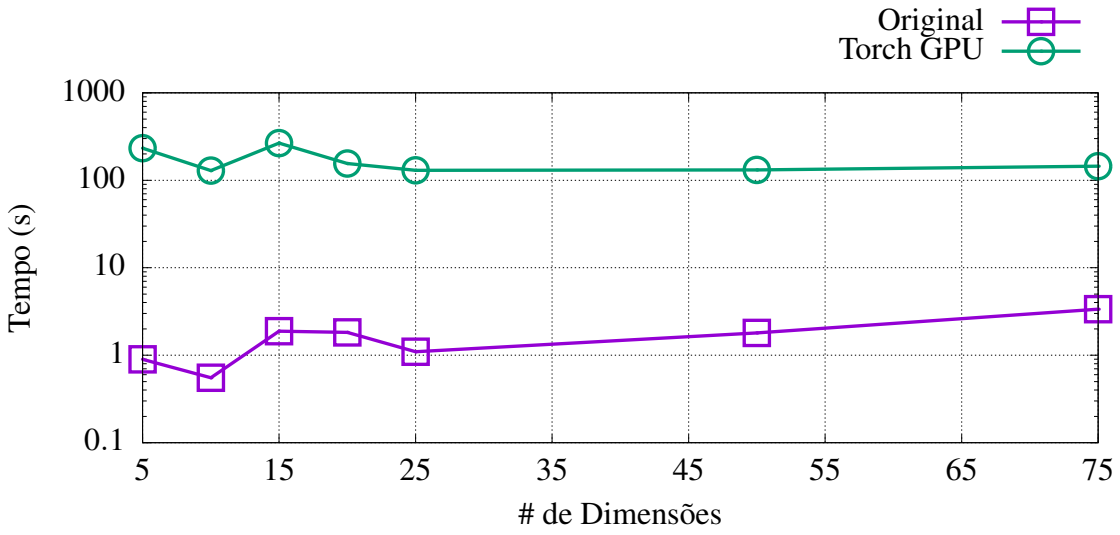

Figura 4.4 Tempo médio de execução vs Número de dimensões.

cer mais devagar, muito menos na maior base, com 5500 padrões. O tempo de execução do algoritmo é linear na quantidade de padrões de entrada, pois ela só afeta o número de épocas de treinamento e o número de competições na fase final de agrupamento. As épocas de treinamento não são paralelizáveis, então não se deve esperar vantagem para a GPU ao aumentar o número de padrões.

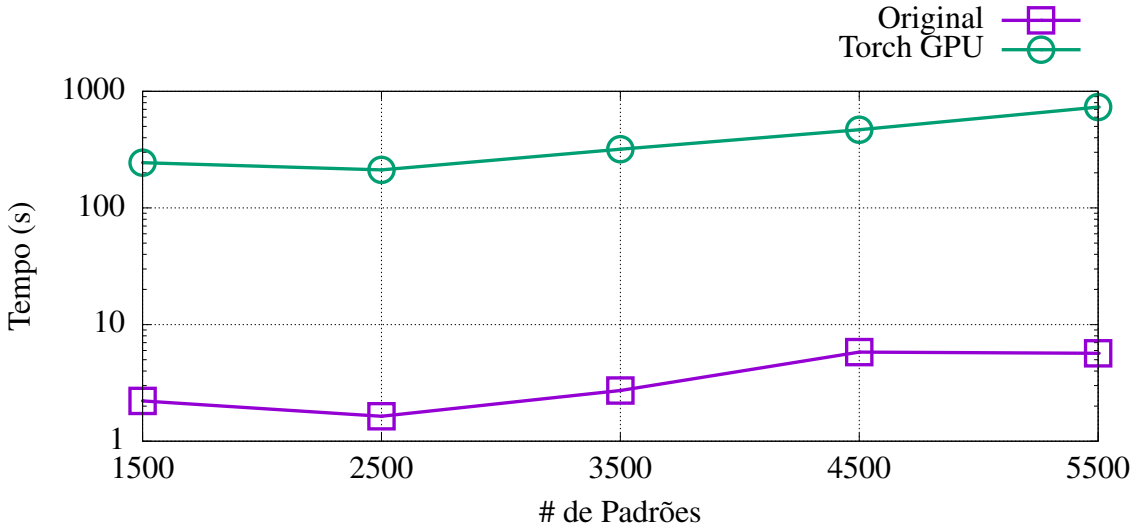

Figura 4.5 Tempo médio de execução vs Tamanho da base de dados.

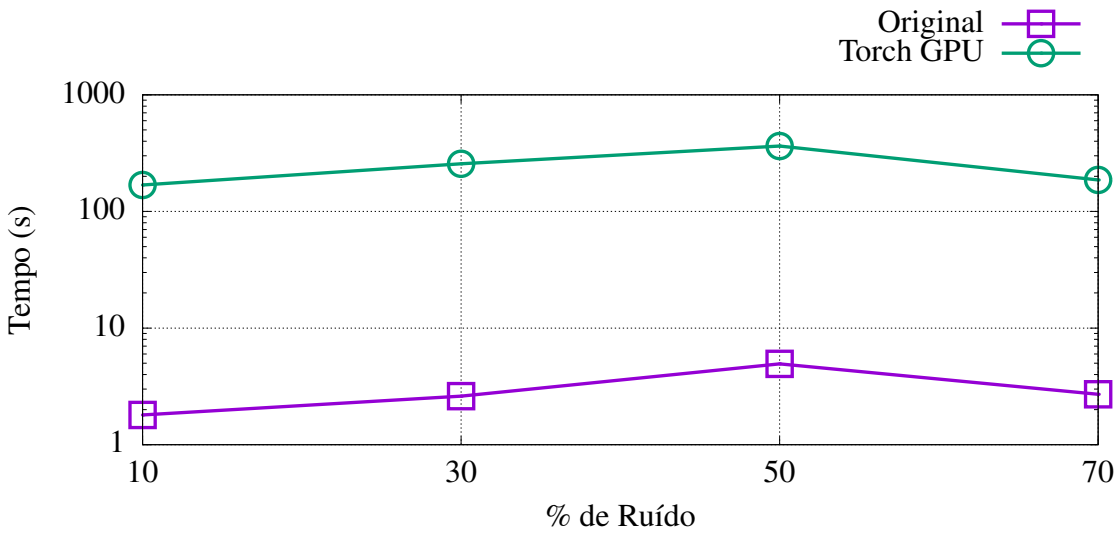

Figura 4.6 Tempo médio de execução vs Porcentagem de ruído nos dados.

4.3 Testes com Bases de Dados Adicionais

O quarto conjunto de bases de dados foi criado para avaliar se a implementação em GPU obteria vantagem com uma quantidade maior de neurônios na rede. Como esse valor não pode ser controlado diretamente, foi variado o número de grupos reais nas bases de dados. Já as bases do quinto conjunto variam exponencialmente em número de dimensões, com o objetivo de tornar mais evidente o ganho de desempenho em bases de dados com muitas dimensões.

Cada base de dados foi criada com 5120 padrões, com os valores dos atributos gerados aleatoriamente no intervalo [0, 1]. Em cada base, cada padrão foi atribuído a exatamente um grupo, então o LARFDSSOM foi executado em modo de agrupamento projetivo e sem filtro de ruído. Em cada execução do algoritmo, o parâmetro N_{max} recebeu como valor a quantidade correta de grupos da respectiva base de dados. Todas as bases do quarto conjunto foram geradas com 32 dimensões, e todas as do quinto conjunto foram geradas com 8 grupos reais. As Tabelas 4.2 e 4.3 mostram os detalhes de cada base de dados.

Todos os grupos em cada base possuíam o mesmo tamanho, e foram gerados em formato de "tubos" como na Figura 4.7: cada grupo possuía exatamente uma dimensão irrelevante, e os valores para cada uma das outras dimensões foram gerados em um intervalo estreito, definido independentemente para cada dimensão.

4.3.1 Qualidade do Agrupamento

As Figuras 4.8 e 4.9 comparam os valores médios de erro de agrupamento obtidos pelas duas implementações. Novamente a implementação em GPU reproduziu a qualidade dos resultados da implementação original. Nas bases do conjunto 4, os altos valores de CE obtidos com baixos desvios padrão são uma evidência de que ambas as implementações do algoritmo estão obtendo a quantidade desejada de neurônios na rede. Mas para confirmar, para cada base foi conferida

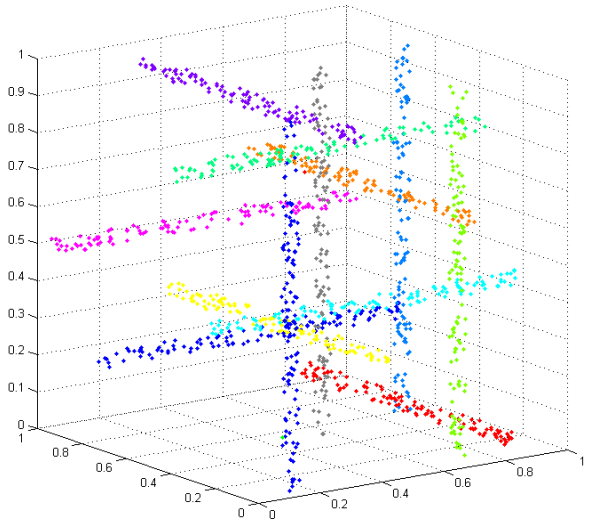

Figura 4.7 Exemplo de base de dados sintética para agrupamento em subespaço com grupos em formato de tubos. Padrões são representados por pontos, e os grupos são indicados pelas diferentes cores. Cada grupo se extende por uma dimensão e se concentra em cada uma das outras.
Fonte: [12]

Tabela 4.2 Propriedades das bases de dados do conjunto 4.

# de padrões	# de dimensões	# de grupos reais
5120	32	8
5120	32	16
5120	32	32
5120	32	64
5120	32	128
5120	32	256

a quantidade de grupos encontrados na última das 30 execuções de cada implementação: em apenas três casos, a quantidade de grupos encontrados estava inferior à correta, e a diferença era de no máximo três grupos.

Nas bases de dados do conjunto 5, percebe-se que o valor de CE obtido cresce junto com o número de dimensões da base, se aproximando do valor máximo 1. Isso acontece porque um número maior de dimensões deixa os grupos mais evidentes e fáceis de distinguir.

4.3.2 Tempo de Execução

As Figuras 4.10 e 4.11 mostram os tempos médios de execução das duas implementações. Não foi identificada nenhuma tendência em especial ao se comparar, na Figura 4.10, os desempenhos em função do número de grupos reais nas bases de dados.

Na Figura 4.11, como esperado, o tempo de execução da implementação original cresce linearmente em função do número de dimensões nas bases. Já o da implementação em GPU permanece quase constante até 4096 dimensões, e a partir daí começa a crescer mais devagar

Tabela 4.3 Propriedades das bases de dados do conjunto 5.

# de padrões	# de dimensões	# de grupos reais
5120	32	8
5120	64	8
5120	128	8
5120	256	8
5120	512	8
5120	1024	8
5120	2048	8
5120	4096	8
5120	8192	8
5120	16384	8

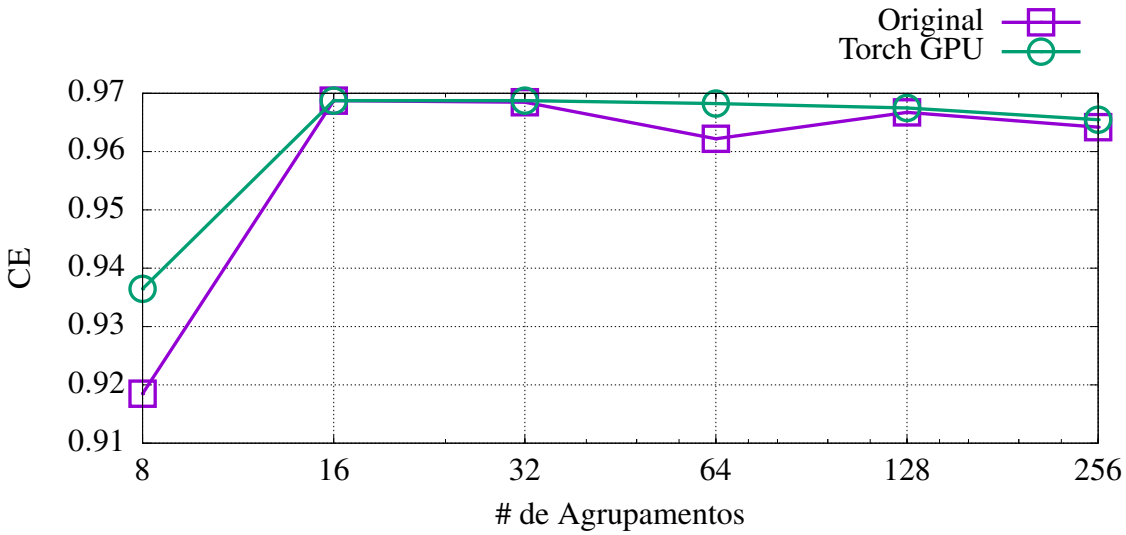

Figura 4.8 Clustering Error vs Quantidade de agrupamentos nos dados.

do que o tempo da implementação original. Isso é uma boa evidência de que o processamento paralelo de GPU está mitigando o efeito do número de dimensões no tempo de execução, ao permitir que o algoritmo manipule paralelamente cada dimensão dos dados. A partir de 8192 dimensões, a implementação em GPU supera o desempenho da original, com um tempo de execução 5% menor. Com 16384 dimensões, essa diferença chegou a 32%.

Neste Capítulo, foram apresentados os testes para avaliar os ganhos de desempenho obtidos com a implementação do algoritmo LARFDSSOM em GPU, bem como os resultados desses testes. No próximo Capítulo, serão traçadas conclusões finais sobre este trabalho.

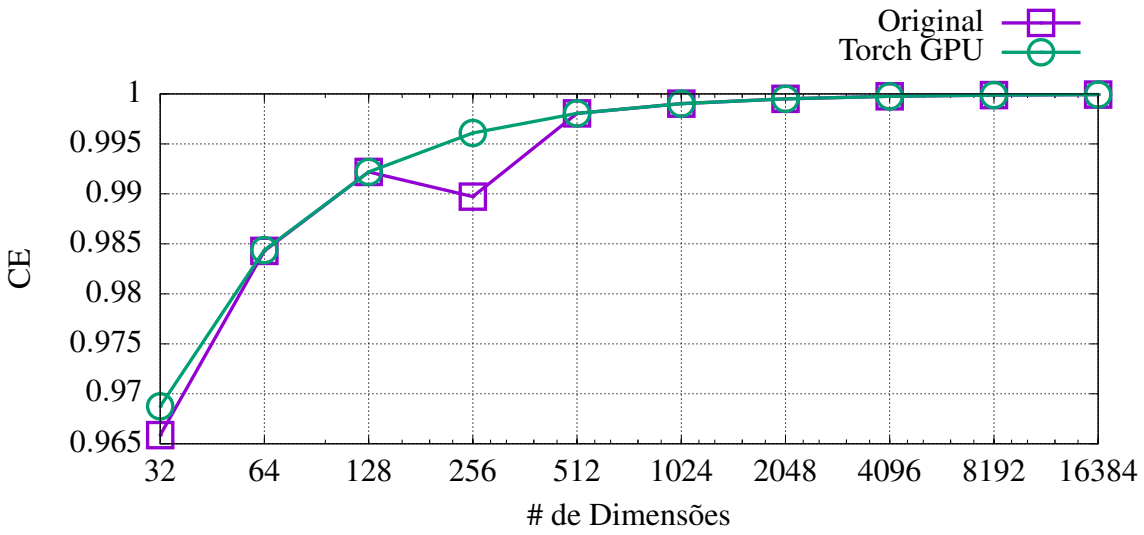

Figura 4.9 Clustering Error vs Número de dimensões.

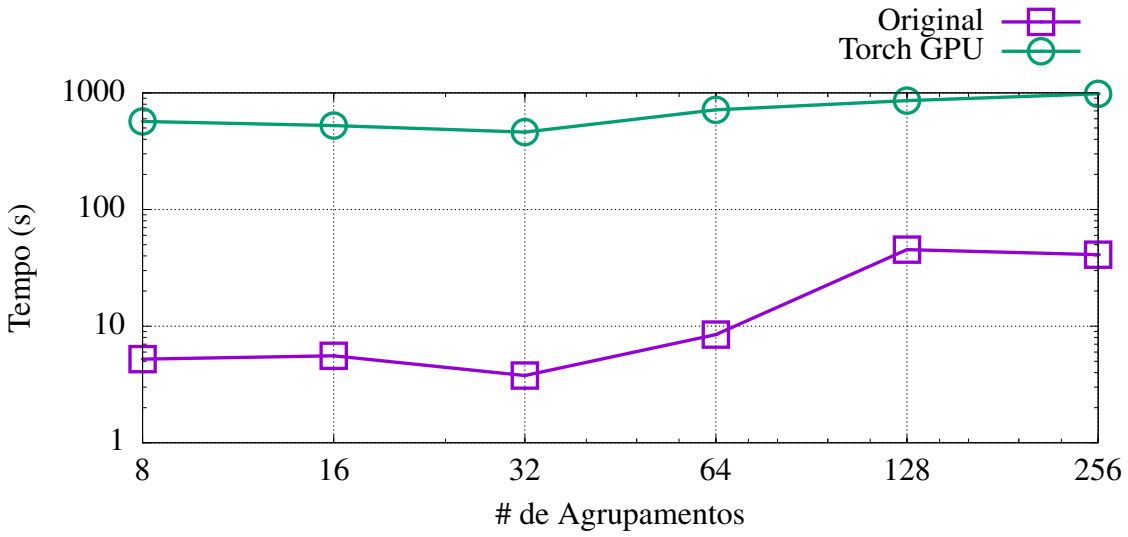

Figura 4.10 Tempo médio de execução vs Quantidade de agrupamentos nos dados.

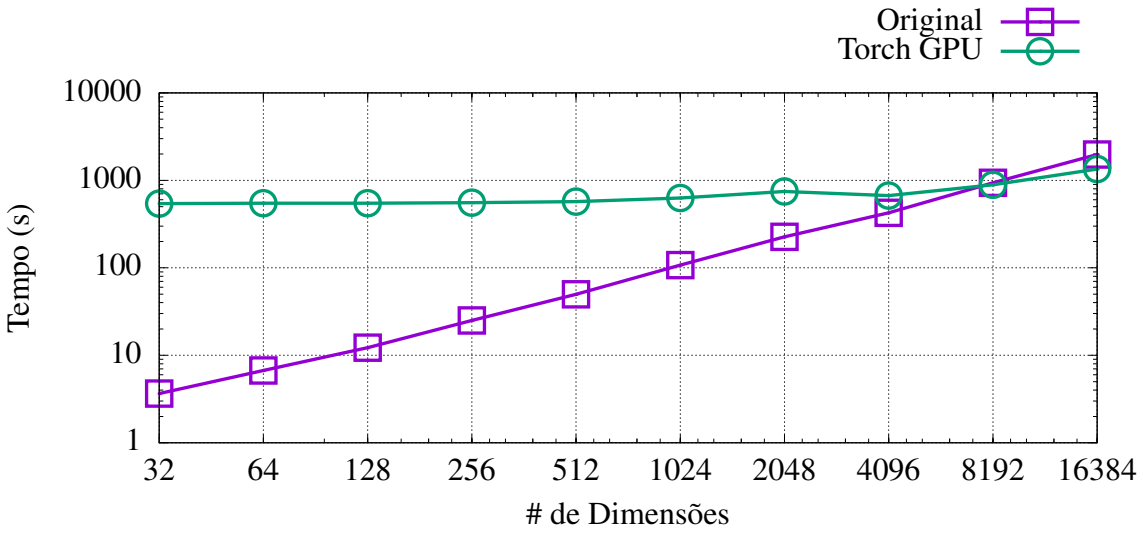

Figura 4.11 Tempo médio de execução vs Número de dimensões.

CAPÍTULO 5

Conclusão

Neste trabalho, foi descrita a implementação em GPU do algoritmo LARFDSSOM para agrupamento em subespaço. Os resultados apresentados no Capítulo anterior mostraram ganhos de desempenho ao se lidar com bases de dados com milhares de dimensões, em comparação com a implementação original em C++. No mundo real é comum realizar agrupamento em bases de dados com dezenas ou centenas de milhares de dimensões, então realizar agrupamento em subespaço com GPUs pode ser vantajoso em situações de alta dimensionalidade, e com grandes quantidades de agrupamentos. Em contrapartida, o tamanho da base de dados ou a proporção de ruído nos dados não mostraram efeito significativo na diferença de desempenho entre as implementações original e em GPU.

O *framework* Torch foi escolhido para a implementação do LARFDSSOM por sua robustez e popularidade, mas principalmente pela facilidade que ele provê para se lidar com operações com tensores paralelizadas em GPU. Entretanto, a implementação desenvolvida neste trabalho não mostrou um desempenho adequado com redes muito pequenas, ou seja, com baixa dimensionalidade ou poucos agrupamentos. Trabalhos futuros podem avaliar a reimplementação do LARFDSSOM ou de outros algoritmos de agrupamento utilizando outras ferramentas mais eficientes, ou em código nativo para GPU. Futuros ganhos de desempenho, devido a implementações mais eficientes ou ao uso de *hardware* mais potente, podem reduzir o *overhead* de execução introduzido tanto pelo uso de dispositivos de GPU quanto pelas ferramentas e *frameworks* utilizados. Isto tornaria o uso de GPU vantajoso também para bases de dados mais modestas, com até mil dimensões, por exemplo.

O algoritmo de treinamento do LARFDSSOM é constituído por uma série de competições por padrões e remoções de neurônios. Para se obter ainda mais ganhos de desempenho com paralelismo, pode-se estudar a possibilidade de se alterar o treinamento para realizar competições simultâneas. A criação de novos neurônios poderia também ser paralelizada. Embora o procedimento de atualização dos vencedores e suas vizinhanças possa não ter efeito equivalente ao atual, baixas taxas de aprendizagem no algoritmo podem mitigar a diferença. Uma implementação em GPU com competições paralelas poderia obter ganhos de desempenho com bases de dados com muitos padrões. Entretanto, são necessários estudos e testes para se certificar de que a qualidade do agrupamento produzido pelo algoritmo não seria sacrificada.

É importante frisar também que o algoritmo avaliado demonstrou ótima escalabilidade com o crescimento do numero de dimensões e do número de agrupamentos nos dados, se mostrando adequado para problemas de larga escala, algo que não havia sido avaliado ainda na literatura.

Referências Bibliográficas

[1] T. M. Mitchell, *Machine Learning*. New York, NY, USA: McGraw-Hill, Inc., 1 ed., 1997.

[2] E. Müller, S. Günnemann, I. Assent, and T. Seidl, "Evaluating clustering in subspace projections of high dimensional data," in *Proc. 35th International Conference on Very Large Data Bases (VLDB 2009), Lyon, France, PVLDB Journal, Vol. 2, No. 1,*, pp. 1270–1281, VLDB Endowment, 2009.

[3] E. M. Smith, J. Littrell, and M. Olivier, "Automated SNP genotype clustering algorithm to improve data completeness in high-throughput SNP genotyping datasets from custom arrays," *Genomics, Proteomics & Bioinformatics*, vol. 5, no. 3–4, pp. 256 – 259, 2007.

[4] R. Vidal, "Subspace clustering," *Signal Processing Magazine, IEEE*, vol. 28, no. 2, pp. 52–68, 2011.

[5] A. Sharma and R. Dhir, "A wordsets based document clustering algorithm for large datasets," in *2009 Proceeding of International Conference on Methods and Models in Computer Science (ICM2CS)*, pp. 1–7, Dec 2009.

[6] H. F. Bassani and A. F. R. Araújo, "Dimension selective self-organizing maps with time-varying structure for subspace and projected clustering," *IEEE Transactions on Neural Networks and Learning Systems*, vol. 26, pp. 458–471, March 2015.

[7] S. Bahrampour, N. Ramakrishnan, L. Schott, and M. Shah, "Comparative study of caffe, neon, theano, and torch for deep learning," *CoRR*, vol. abs/1511.06435, 2015.

[8] T. Kohonen, "The self-organizing map," *Proceedings of the IEEE*, vol. 78, pp. 1464–1480, Sep 1990.

[9] H. F. Bassani and A. F. R. Araújo, "Dimension selective self-organizing maps for clustering high dimensional data," in *The 2012 International Joint Conference on Neural Networks (IJCNN)*, pp. 1–8, June 2012.

[10] "Torch's main website." `http://torch.ch`. [Online; accessed 30-June-2017].

[11] J. Helton, F. Davis, and J. Johnson, "A comparison of uncertainty and sensitivity analysis results obtained with random and latin hypercube sampling," *Reliability Engineering & System Safety*, vol. 89, no. 3, pp. 305 – 330, 2005.

[12] H. F. Bassani, *Modelos Neurais de Aquisição de Linguagem Natural para Agentes Incorporados*. PhD thesis, Universidade Federal de Pernambuco, 2014.

[13] E. P. Macedo, "Agrupamento em subespaço com mapas auto-organizáveis em gpu," Dec. 2016.

Este volume foi tipografado em LaTeX na classe UFPEThesis (www.cin.ufpe.br/~paguso/ufpethesis).