João Pedro Vicente Martins Borrego

Licenciatura em Ciência e Engenharia Informática

Mapas Auto-Organizados Ubíquos em Unidades de Processamento Gráfico

Dissertação para obtenção do Grau de Mestre em

Engenharia Informática

Orientadores: Hervé Miguel Cordeiro Paulino, Professor Auxiliar, Universidade Nova de Lisboa

Nuno Miguel Cavalheiro Marques, Professor Auxiliar, Universidade Nova de Lisboa

Júri

Presidente:	Name of the male committee chairperson
Arguentes:	Name of a female raporteur
	Name of another (male) raporteur
Vogais:	Another member of the committee
	Yet another member of the committee

FACULDADE DE CIÊNCIAS E TECNOLOGIA
UNIVERSIDADE NOVA DE LISBOA

Setembro, 2018

Mapas Auto-Organizados Ubíquos em Unidades de Processamento Gráfico

Este documento foi gerado utilizando o processador (pdf)LaTeX, com base no template "novathesis" [1] desenvolvido no Dep. Informática da FCT-NOVA [2].
[1] https://github.com/joaomlourenco/novathesis [2] http://www.di.fct.unl.pt

Resumo

Atualmente, quantidades gigantescas de informação são geradas a cada instante. Tecnologias como a computação ubíqua (*Internet Of Things*), as redes sociais e outras fontes geram informação constantemente. Existe a necessidade de processar essa informação para auxiliar a tomada de decisões, muitas vezes em tempo real. Para satisfazer essa necessidade, vários algoritmos têm sido idealizados com o objetivo de analisar sequências de dados e reconhecer padrões existentes nessas sequências de dados. Um desses algoritmos é o Mapa Auto-Organizado Ubíquo, ou UbiSOM, uma variante do Mapa Auto-Organizado (SOM) que permite criar um modelo com base em sequências de dados potencialmente infinitas. No entanto, o CPU pode ter dificuldade em processar sequências de dados em tempo real. Muitos algoritmos de Aprendizagem Automática utilizam a Unidade de Processamento Gráfico (GPU), para acelerar o ritmo em que as computações são efetuadas. Devido às várias oportunidades de paralelização existentes, o algoritmo UbiSOM é um bom candidato para receber uma implementação em GPU, com o potencial para aumentar o ritmo de processamento do algoritmo, com as devidas otimizações.

Para esta dissertação, implementou-se uma versão do UbiSOM adaptada para execução em GPU, utilizando a ferramenta Marrow, uma ferramenta para desenvolvimento de sistemas heterogéneos para C++, utilizando construções de alto nível, com o objetivo de permitir ao programador focar-se no aspeto lógico da implementação e menos nos detalhes de cada dispositivo. A utilização dessa ferramenta tem como objetivo verificar a utilidade da mesma para aplicações reais e permitir o amadurecimento dela.

Nesta dissertação será discutida a implementação do algoritmo e de todo o sistema à volta desse algoritmo. Será debatido ainda o trabalho efetuado em melhorar o Marrow. Serão também discutidos os resultados obtidos em termos de *performance*, comparando execuções em CPU com execução aceleradas por GPU. Por fim vai-se comparar o comportamento do UbiSOM adaptado para GPU com o comportamento do UbiSOM original.

Palavras-chave: UbiSOM, GPU, Processamento de *Streams*, Computação Heterogénea, Marrow

Abstract

In our days, massive amounts of information is generated at each instant. Technologies like pervasive computing (Internet of Things), social networks and other sources produce information constantly. There is a need to process that information in order to aid decision taking, many times in real-time. To satisfy that need, several algorithms have been devised with the goal of analysing those data streams and recognizing patterns in those streams. One of those algorithms is the Ubiquitous Self-Organizing Map, or UbiSOM, a variant of the Self-Organizing Map (SOM), that allows us to generate a model from potentially infinite data streams. However, the CPU may have difficulty in processing data streams in real-time. Many Machine Learning algorithms already use an additional source of processing power to accelerate the calculations: the Graphic Processing Unit (GPU). With this in mind, and due to the several parallelization opportunities present in the algorithm, UbiSOM is a great candidate to receive a GPU implementation, having the potential of accelerating the algorithm with the applicable optimizations.

For this dissertation, a GPU-oriented implementation of UbiSOM was made, using the Marrow Framework, a tool to develop heterogeneous systems in C++. This tool aims to allow the developer to focus on the logical aspect of the implementation, leaving the device details to the framework. The goal of using this tool is to verify how useful it is when implementing real life applications, and to allow its maturing.

During this dissertation it will be discussed the implementation of the algorithm and the system surrounding it. It will also be discussed the work made in order to improve the Marrow *framework*. Then we will compare the performance of the implementation with a CPU-only implementation. At last we will compare the behaviour of the GPU-optimized UbiSOM with the behaviour of the original UbiSOM.

Keywords: UbiSOM, GPU, Stream Processing, Heterogeneous Computing, Marrow

Índice

Lista de Figuras

Listagens

Glossário

kernel	Função declarada num programa e executada num dispositivo OpenCL.
MapReduce	Modelo de programação para processar conjuntos de dados num ambiente paralelo ou distribuído. Consiste numa operação de mapeamento, aplicando uma determinada função a cada elemento de um conjunto de dados, e numa operação de redução, para sumarizar o resultado da operação de mapeamento.
SIMD	*Single Instruction Multiple Data*. Todas as unidades executam o mesmo programa sobre múltiplos segmentos de dados de forma sincronizada.
SPMD	*Single Program Multiple Data*. Todas as unidades executam o mesmo programa sobre múltiplos segmentos, mas de forma independente.
texture binding	Método utilizado em Computação Gráfica para mapear dados frequentemente utilizados pelo GPU.

Capítulo 1

Introdução

1.1 Motivação

Atualmente, quantidades gigantescas de informação são criadas a cada instante. Tecnologias como computação ubíqua (*Internet Of Things*), redes sociais e muitas outras geram quantidades de informação sem parar, existindo a necessidade de processar essa informação de forma a extrair conhecimento para ser utilizado na tomada de decisões. Podemos encontrar exemplos na área financeira [38, 42], em aplicações web [4], redes de computadores [80] ou monitorização de sensores [21, 57].

A computação ubíqua, por seu lado, inclui dispositivos como sensores inteligentes, sistemas de controlo de veículos e eletrodomésticos, periféricos de computadores e outros sistemas de computadores embebidos que produzem quantidades massivas de dados, usualmente designados como fontes de dados ubíquas. A necessidade de analisar esta quantidade enorme de dados deu origem ao campo de *Ubiquitous Data Mining* [20], um campo emergente que se dedica a descoberta de conhecimento a partir de informações obtidas a partir de dispositivos móveis, embebidos e ubiquos e das redes de processamento por eles formados.

Exemplos das aplicações anteriormente mencionadas são:

- Agrupamento de sequências de dados obtidos a partir de sensores ou redes de sensores [21, 57, 64], obtendo, p.e., medições da concentração de poluentes numa cidade ou monitorização de sensores automóveis [39].

- Aplicações financeiras [38] exigem conhecimento em tempo-real, uma vez que ordens de negociação são colocadas em cada segundo, com variações continuas nos preços de um determinado conjunto de *assets* financeiros. A agregação destes dados pode permitir identificar quais desses *assets* são similares ou distintos, auxiliando a

gestão de portefólios e a tomada de decisões na altura de colocar ordens de negociação.

- Recolha de informações em ambientes de computação móvel ou distribuída [71], permitindo que dados gerados por dispositivos como *smartphones* sejam utilizados para tomada de decisões.

Ser capaz de produzir modelos em tempo real é de grande importância para essas aplicações. No entanto, o facto de essas informações chegarem em tempo real (ou quase), com um grande volume (potencialmente infinito) e com imensa variedade, fazem com que seja impraticável guardar essas informações de forma permanente [5]. Algoritmos têm sido desenvolvidos com o objetivo de manipular esse fluxo de dados para que não seja necessário armazenar os dados em questão [22].

Algoritmos de inteligência artificial e aprendizagem automática foram idealizados, com o objetivo de ajudar a encontrar padrões escondidos dentro dessa quantidade infinita de informações. Entre esses algoritmos está o SOM, o Mapa Auto-Organizado, uma ferramenta muito utilizada para projetar dados com muitas dimensões para um mapa bidimensional, preservando as relações entre os dados de entrada. O SOM permite analisar facilmente a estrutura dos dados processados através de várias técnicas de visualização disponíveis.

No entanto, apesar destas vantagens, o SOM original não pode ser aplicado a fontes de dados infinitas, uma vez que o SOM requer o conhecimento do número de observações *à priori*. O UbiSOM, o Mapa Auto-Organizado Ubíquo [68], possui adaptações que dão ao mapa uma maior flexibilidade em lidar com fontes de dados potencialmente infinitas.

É de considerar porém, que devido à natureza das fontes de dados, e devido ao requisito de tempo real que normalmente as acompanha, a velocidade de processamento é fundamental. Um método muito utilizado é o aproveitamento do poder de processamento de Unidades de Processamento Gráfico (GPU) para executar esse processamento [14, 52, 55, 77, 79]. Ao utilizarmos CPUs e GPUs para executar algoritmos, entramos assim na área da computação heterogénea [31]. No entanto, o desenvolvimento de sistemas heterogéneos envolve conhecimento intrínseco da arquitetura alvo. Com isso em mente, *frameworks* de mais alto nível têm surgido. Uma dessas *frameworks* é o Marrow [16], cujo objetivo é permitir o desenvolvimento de sistemas heterogéneos, focando no aspeto lógico da aplicação e deixando coisas como a otimização no lado da *framework*.

1.2 Problema

Com o UbiSOM disponível, temos uma ferramenta para mapear dados em *streams*. No entanto, as implementações atuais do UbiSOM fazem o processamento em CPU, o que para conjuntos de informação vastos, ou para mapas grandes, não é apropriado, podendo até prejudicar o requisito de *real-time* presente em aplicações de processamento de *streams*, devido ao elevado requisito computacional presente no SOM (e variantes).

Uma solução para este problema está num componente cada vez mais presente nos dispositivos atuais: a Unidade de Processamento Gráfico, ou GPU, cuja utilização tem sido generalizada na área da aprendizagem automática. Algoritmos bastante utilizados como o *K-Means* e até o próprio SOM e algumas das suas variantes, têm sido acelerados com a ajuda de GPUs, aumentando a velocidade com que a aprendizagem é efetuada [14, 52, 55, 77, 79].

Com isto em mente, foi proposto adaptar o UbiSOM para aproveitar o poder de processamento disponível através dos GPUs. Para isso é fundamental investigar o processamento de *streams* através de GPUs e como executar a implementação do UbiSOM em GPUs.

Um aspeto importante é o aproveitamento do poder de processamento do GPU. A biblioteca Marrow, desenvolvida na secção 2.2.3, permite-nos focar no aspeto lógico da implementação. No entanto, será ainda necessário expandir o Marrow para satisfazer os requisitos do UbiSOM, uma vez que devido ao facto desta biblioteca ainda estar em desenvolvimento, poderão ser necessárias funcionalidades que ainda não estão implementadas.

O GPU, além de permitir a paralelização do processamento de uma observação, permite ainda o processamento paralelo de múltiplas observações. No entanto, tal abordagem pode ter um efeito sobre a convergência do UbiSOM ao paralelizar o algoritmo por processar várias observações ou unidades em simultâneo. Existe um precedente entre o Online-SOM e o Batch-SOM, uma versão do SOM que em vez de aplicar uma observação de cada vez, aplica várias observações [41], em que se observa resultados divergentes entre os dois algoritmos, nomeadamente a dificuldade do Batch-SOM em organizar a informação, mostrando que alterações na forma como a informação é processada resulta em alterações no resultado final[17].

1.3 Solução

Com os objetivos mencionados na secção 1.2 em mente propõe-se implementar o sistema ilustrado na figura 1.1. Como é possível analisar na figura, o sistema baseia-se numa *pipeline* que numa extremidade recebe a *stream* de dados e envia esses dados para serem processados pelo UbiSOM, posteriormente permitindo a visualização do modelo.

Ao receber a fonte de dados, começa-se por discretizá-la. Para isso, existem duas opções: ou discretiza-se cada observação individualmente, enviando-a de seguida ao GPU, ou agrupa-se as diferentes observações em *mini-batches*, permitindo enviar várias observações de cada vez ao GPU.

Ao executar o UbiSOM no GPU, existe a hipótese de paralelizar várias etapas do processamento do UbiSOM de duas maneiras: por implementar uma *pipeline* (na figura: k_1, k_2 e k_3) em que cada observação é processada individualmente em cada etapa da *pipeline*, e por potencialmente, executar duas ou mais instâncias da *pipeline* em paralelo (na figura: Ubisom 1 e 2).

Para implementar o UbiSOM em GPUs, é proposto utilizar a linguagem de programação C++ e a biblioteca Marrow para desenvolver os *kernels* para submeter ao GPU e

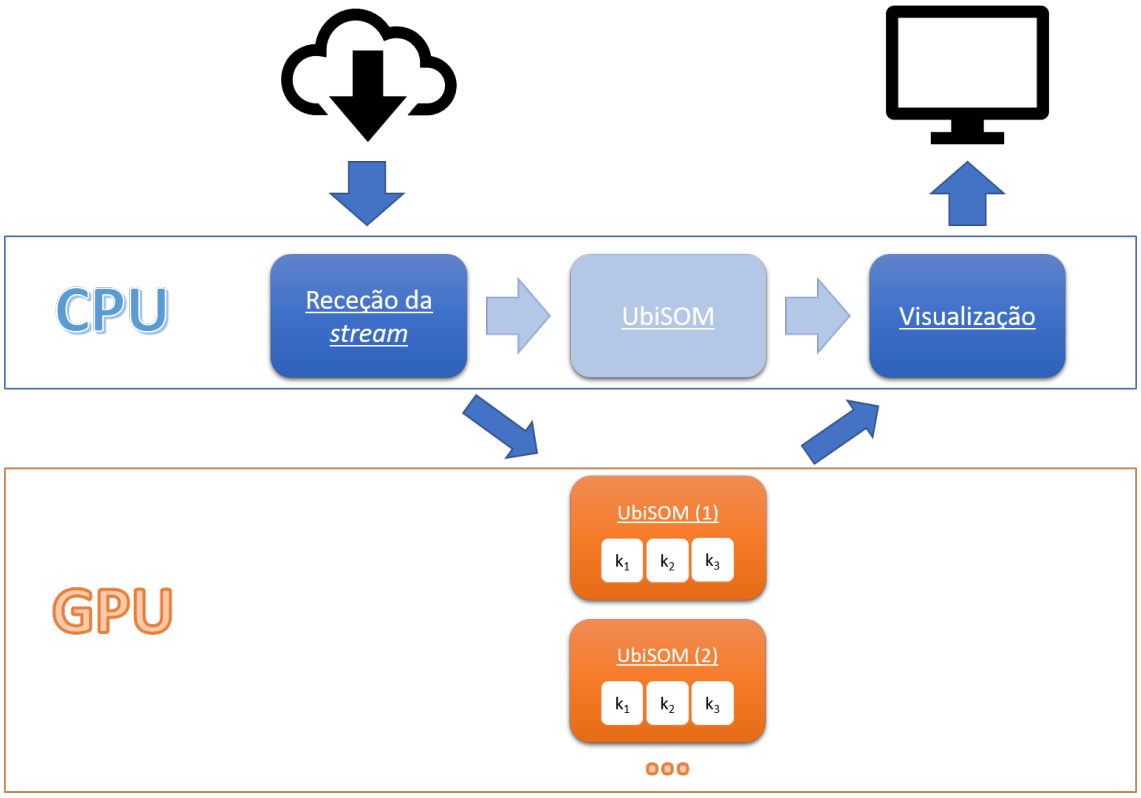


Figura 1.1: Diagrama de Sistema Simplificado

ligar a execução desses *kernels*, isto numa linguagem de alto nível em comparação com o OpenCL. Para além disso, tem-se o objetivo de auxiliar o amadurecimento da biblioteca Marrow enquanto se desenvolve um exemplo da vida real que é relevante.

1.4 Contribuições

O objetivo desta tese é acelerar o processamento do UbiSOM com a utilização de GPUs, utilizando a *framework* Marrow, com o objetivo de permitir ao UbiSOM processar quantidades superiores de informação. Posteriormente proceder-se-á à validação dos ganhos de desempenho de forma experimental.

Irá ser também demonstrado a praticabilidade do Marrow em implementações do mundo real para GPUs, e permitir e amadurecimento da plataforma com novas funcionalidades.

1.5 Estrutura do Documento

Este documento encontra-se dividido em 5 capítulos, distintos, cada um com as suas subsecções.

Este primeiro capítulo consiste na introdução ao documento, onde se elabora a motivação por trás do trabalho realizado, qual o problema a resolver e um sumário inicial da arquitetura da solução implementada.

O segundo capítulo consiste na revisão de literatura, onde é analisado o trabalho relevante já efetuado para a solução do problema já mencionado no primeiro capitulo. Começa-se por analisar algoritmos conhecidos de Análise Exploratória de Dados, como por exemplo, o *K-Means*, o Mapa Auto-Organizado, e o Mapa Auto-Organizado Ubíquo, o algoritmo que serve de base para este projeto. De seguida, discute-se as Unidades de Processamento Gráfico, a sua arquitetura e métodos de desenvolver programas para correr em GPUs. Também se provê uma análise do modelo de plataforma, execução e programação do OpenCL, juntamente com uma análise do Marrow. Por fim, termina-se o capitulo por analisar o trabalho efetuado na implementação de Mapas Auto-Organizados em GPUs.

No terceiro capítulo discute-se a arquitetura utilizada na implementação do servidor UbiSOM, discutindo o uso do modelo cliente-servidor e de *pipelines* para maximizar o paralelismo de tarefas e aborda-se alguns detalhes sobre a implementação do algoritmo UbiSOM, o trabalho efetuado sobre o Marrow, e de como a comunicação com o servidor é efetuada.

No quarto capítulo, são elaborados os diversos testes para validar a implementação do UbiSOM em GPU. Começa-se por discutir a metodologia desses testes. De seguida, discute-se os testes unitários implementados para validar a implementação do UbiSOM. Faz-se de seguida a comparação do desempenho do UbiSOM entre execuções em CPU e com GPU, utilizando o *dataset* Irís e algumas variantes. Por fim, utiliza-se várias fontes de dados utilizadas para testar o UbiSOM original para comparar as duas implementações.

No quinto e último capítulo, faz-se algumas considerações finais sobre o trabalho efetuado, e propõe-se trabalho futuro para ser executado.

Finalmente, existe as secções de apêndices e anexos, contendo vários excertos do *software* implementado durante esta dissertação.

Capítulo 2

Revisão de Literatura

Tendo em conta o tema do projeto em questão, é necessário obter um entendimento fundamental do funcionamento dos Mapas Auto-Organizados (SOM), nomeadamente do algoritmo para o Mapa-Organizado Ubíquo, e de onde é que este modelo se origina. Assim, este capítulo vai-se iniciar por analisar algoritmos como o *K-Médias*, um dos primeiros algoritmos de agrupamento de dados, o SOM e o UbiSOM.

Outro ponto fundamental é a sua implementação em Unidades de Processamento Gráfico, ou GPU. Assim sendo, vai-se analisar o funcionamento de um GPU e os diferentes modelos de programação disponíveis, com foco principal no Marrow, a ferramenta que vai ser utilizada neste projeto, e no OpenCL, a biblioteca de baixo nível que serve de base para o Marrow.

Por fim, vai-se analisar o trabalho efetuado anteriormente com implementações de SOMs em GPUs, e verificar quais as implementações e otimizações efetuadas.

2.1 Algoritmos para Análise Exploratória Avançada de Dados

Análise Exploratória de Dados (AED) é definido por John Tukey [73] como sendo o conjunto de procedimentos para analisar dados, de técnicas para interpretar os resultados desses procedimentos, os métodos de planear a recolha de dados para facilitar a sua análise e toda a maquinaria e resultados estatísticos aplicados à analise de dados.

AED tem o objetivo de sugerir hipóteses acerca das causas de fenómenos observados, definir as premissas a partir das quais se baseia a inferência estatística, suportar a seleção apropriada de ferramentas e técnicas e prover uma base para recolhas de dados posteriores [6].

Nesta secção, iremos analisar vários algoritmos que suportam técnicas avançadas de

AED para obter um entendimento base do seu funcionamento, como por exemplo, o *K-Médias* e o Mapa Auto-Organizado (SOM), com um maior foco no Mapa Auto-Organizado. Também iremos analisar uma variante do SOM, o Mapa Auto-Organizado Ubíquo (Ubi-SOM).

2.1.1 Agrupamento K-Médias

O método de agrupamento K-Médias [29] é um método de quantização de vetores, muito popular em *data mining*. Este método tem como objetivo particionar n observações em k *clusters* de forma a que cada observação pertença ao *cluster* com o valor médio mais próximo, sendo este valor considerado o protótipo (ou centróide) do agregamento.

2.1.1.1 Algoritmo

O algoritmo de agrupamento K-Médias, também chamado de algoritmo de Lloyd [44], utiliza uma técnica de refinamento iterativo, alternando entre fases distintas: atribuição e atualização [48].

Inicialização Quanto à inicialização do K-Médias, existem dois métodos bastante utilizados: o método de Forgy e o método da partição aleatória. O método de Forgy escolhe aleatoriamente k observações e utiliza esses pontos como valores médios iniciais. O método de partição aleatória começa por atribuir de forma aleatória um agrupamento a cada observação, e depois executa a fase de atualização para calcular os centróides dos *clusters* gerados aleatoriamente [25].

Fase de Atribuição Nesta fase, atribui-se cada observação ao agrupamento cujo valor médio possua a menor distância euclidiana, ou seja ao protótipo mais próximo, como observado no algoritmo 1.

Algorithm 1 Fase de Atribuição do K-Médias

```
means ← centróides de cada agrupamento.
sets ← agrupamentos de observações
k ← numero de agrupamentos.
procedure ATRIBUTION_PHASE(observations)
    for all obs ∈ observations do
        distances ← ∀_{0<i<k}(|obs − means[i]|^2)
        closest ← argmin(distances)
        sets[closest] ← sets[closest] ∪ obs
    end for
end procedure
```

O algoritmo 1 demonstra a fase de atribuição do K-Médias. Para cada observação, calcula qual dos centróides é mais próximo da observação, e depois atribui a observação ao conjunto correspondente.

Fase de Atualização Calcula o novo valor médio, ou centróide, para o novo conjunto de observações.

Algorithm 2 Fase de Atualização do K-Médias

```
1: means ← centróides de cada agrupamento.
2: sets ← agrupamentos de observações
3: k ← numero de agrupamentos.
4: procedure UPDATE_PHASE
5:     for i = 0 to k do
6:         means[i] ← origin vector
7:         for all obs ∈ sets[i] do
8:             means[i] ← means[i] + obs / |sets[i]|
9:         end for
10:    end for
11: end procedure
```

O algoritmo 2 demonstra-nos a fase de atualização do K-Médias. Para cada *cluster* calcula-se o seu centróide com base nas observações em cada conjunto calculado na fase de atribuição, iterando por cada observação para calcular a sua contribuição para a média.

Convergência Considera-se que o algoritmo convergiu quando não existe nenhuma modificação aos centróides e aos *clusters* [29].

2.1.1.2 K-Médias em GPUs

Farivar et Al. [14] demonstram que o K-Médias tem várias oportunidades de paralelização que podem ser implementadas para executar em GPUs. Os autores argumentam que a fase do algoritmo mais demorada é a fase de atribuição. Eles também argumentam que a paralelização da fase de atualização não tem o mesmo impacto que a paralelização da fase de atribuição, uma vez que o maior ponto de congestão da fase de atualização são os acessos á memória e não o poder de processamento, uma ideia que vemos repetida no trabalho de Zechner e Granitzer [79], que executam a fase de atualização no CPU.

2.1.2 Mapa Auto-Organizado

O Mapa Auto-Organizado (SOM, do inglês *Self-Organizing Map*) é uma forma de Rede Neuronal Artificial (ANN, do inglês *Artificial Neural Network*), com o objetivo de projetar um conjunto de dados multi-dimensional para uma grelha bidimensional de fácil interpretação, normalmente sob a forma de um mapeamento ordenado, ou topologia. Cada nó da grelha é associado com um protótipo (ou unidade), sendo cada protótipo calculado pelo algoritmo SOM. Cada item de dados será então mapeado para o nó mais semelhante, comparado através de uma medição de distância. Assim, nós adjacentes terão protótipos semelhantes.

Tendo sido originalmente concebido por Teuvo Kohonen, é também chamado de **Rede de Kohonen** [41].

2.1.2.1 Algoritmo Online-SOM

O algoritmo Online para Mapas Auto-Organizados (por simplicidade, e sempre que não existir ambiguidade, será referido por SOM) pode ser sub-dividido em várias partes. Nesta secção irá ser discutido cada uma dessas partes, e por fim combinadas sob o algoritmo SOM completo.

Inicialização A inicialização do algoritmo SOM é normalmente efetuada por atribuir vetores aleatórios a todas as unidades (ou protótipos) do mapa. Tal atribuição é feita de forma aleatória, uma vez que, segundo Kohonen [40], ao longo do tempo, estes vetores serão ordenados, formando a topologia do mapa.

Há que salientar que uma inicialização linear do algoritmo também é possível, e segundo investigadores como Akinduke et al. [2] tal inicialização é recomendável para conjuntos de dados de natureza linear.

Taxa de Aprendizagem A Taxa de Aprendizagem [76] é o ritmo a que a aprendizagem do mapa é efetuada, devendo, pela definição, decrescer de forma monótona à medida que o mapa converge.

Segundo Kohonen [40], o período durante o qual um SOM consegue obter uma certa ordem é relativamente curto: aproximadamente 1000 passos. A maior parte do tempo de computação é dedicado á fase final de convergência, com o objetivo de afinar o modelo resultante.

Existem várias funções de vizinhança. No entanto, no caso do SOM, as mais habituais de se usar são a linear (equação 2.1), inversa do tempo (equação 2.2) e séries de potências 2.3. Nas funções mencionadas T representa o número total de iterações, e t a iteração atual [56].

$$\alpha(t) = \alpha(0)\frac{1}{t} \tag{2.1}$$

$$\alpha(t,T) = \alpha(0)\Big(1 - \frac{t}{T}\Big) \tag{2.2}$$

$$\alpha(t,T) = \alpha(0)e^{\frac{1}{T}} \tag{2.3}$$

Vemos no algoritmo 3 uma implementação da taxa de aprendizagem baseado numa função linear, devido à sua simplicidade.

Algorithm 3 Taxa de Aprendizagem

```
t ← iteração atual do algoritmo.
function LEARNING_RATE
    return 1.0/t
end function
```

Best Matching Unit A *Best Matching Unit*, ou BMU, representa a célula mais semelhante à entrada x, como definido na equação 2.4 [40]:

$$\|x - m_c\| = \min_i \|x - m_i\| \tag{2.4}$$

sendo m_c a unidade definida como BMU da observação x atual e m_i uma determinada unidade do mapa. Esta equação demonstra-nos que a unidade selecionada como BMU é a que possuí a menor distância euclidiana entre a observação x e o protótipo m_i.

Função de vizinhança A Função de Vizinhança é a função que define quais as unidades que serão afetadas numa determinada iteração do treino do mapa. Esta função de vizinhança deve ter como centro a BMU em relação ao ponto de entrada atual. Por fim, Kohonen [40] indica que a função de vizinhança deve ser monótona e decrescente.

Ao implementar esta função de vizinhança, queremos, de uma forma geral, aplicar uma interação lateral diretamente, definindo uma vizinhança definida como N_c em volta de uma unidade c, sendo c a posição da unidade definida como BMU. Em cada etapa de aprendizagem todas as células dentro da vizinhança são atualizadas, sendo as restantes células ignoradas. Esta vizinhança é centrada em volta da BMU.

É vantajoso permitir que a vizinhança seja bastante larga no início da aprendizagem e reduzi-la de forma monótona com o tempo. Isto garante que com o passar do tempo, quando associado a uma grande taxa de aprendizagem, o mapa adquira uma certa ordem global. É ainda possível terminal o processo com $N_c = c$, ou seja, atualizar apenas a unidade "vencedora", reduzindo o SOM a um simples modelo de aprendizagem competitiva, mas isto já com a ordem topológica do mapa formada.

Com estas noções em mente, podemos definir a função de vizinhança como visto no algoritmo 4.

Algorithm 4 Função de Vizinhança

```
maximum_distance ← distância máxima inicial entre dois pontos na inicialização do SOM.
t ← iteração atual do algoritmo
function NEIGHBORHOOD_FUNCTION(bmu, current_point, inputs_count)
    theta ← (maximum_distance/2) − ((maximum_distance/2) ∗ (t/inputs_count))
    sqrTheta ← theta^2
    sqrDist ← |bmu − current_point|^2
    return e^(sqrDist/sqrTheta)
end function
```

Este algoritmo começa por considerar o quadrado da distância máxima inicial entre dois pontos do mapa para calcular a distância máxima de pertença a uma vizinhança de acordo com o valor temporal (t) atual. Depois calcula o quadrado da distância entre a BMU e o ponto a ser considerado. Por fim, aplica a divisão desses dois valores como expoente para a função exponencial, devolvendo um valor entre 0 e 1.

Atualização do Mapa Na equação 2.6 vemos a definição da função de atualização do mapa idealizada por Kohonen [40].

$$m_i(t) = [\mu_{i1}(t), \mu_{i2}(t), ..., \mu_{in(t)}] \tag{2.5}$$

$$m_i(t+1) = m_i(t) + \alpha(t) * h_{ci}(t) * [x_i - m_i(t)] \tag{2.6}$$

Podemos ver nessa equação (2.6) o fator $\alpha(t)$, a taxa de aprendizagem para o instante t, e o fator $h_{ci}(t)$, a função de vizinhança para o instante t. Ainda é de notar que x_i é a entrada atual a ser considerada e que m_i é a unidade do mapa a ser afetada atualmente, definida pela equação 2.5, onde μ representa um escalar, e n representa o número de dimensões que o protótipo contém. A expressão $x_i - m_i(t)$ denota o vetor de distância entre x_i e m_i (no instante t), que é depois multiplicado pela taxa de aprendizagem e pela função de vizinhança para obter qual é o vetor que se vai incrementar ao protótipo $m_i(t)$ para obter $m_i(t+1)$.

Algorithm 5 Função de Atualização

1: $map \leftarrow$ mapa do SOM
2: **procedure** UPDATE$(t, i, bmu, input_count)$
3: $\quad learn\_rate \leftarrow learning\_rate()$
4: $\quad neighborhood \leftarrow \forall_k(neighborhood\_function(bmu, k, input\_count))$
5: $\quad map \leftarrow \forall_k(map[k] + (learn\_rate * neighborhood[k] * (i - map[k])))$
6: **end procedure**

No algoritmo 5 obtemos primeiro o cálculo da taxa de aprendizagem para a iteração atual. Se seguida, calculamos os resultados das funções de vizinhança para todas as unidades do mapa. Por fim, usamos a taxa de aprendizagem e o resultado da função de vizinhança para atualizar o mapa em relação ao vetor de entrada atual para cada elemento do mapa.

Algoritmo Agora que temos componentes importantes como a taxa de aprendizagem, a função de vizinhança e a função de atualização definidas, podemos definir o algoritmo completo.

O treino de um Mapa Auto-Organizado é efetuado ao longo de várias iterações, em que em cada iteração é processada uma observação.

Numa fase inicial, calcula-se a distância entre o ponto de treino i atual e todos os pontos do mapa, do qual se extrai a **BMU** por selecionar o ponto do mapa com menor

distância a i. Por cada novo ponto efetuado incrementa-se o valor de t, um índice temporal que representa qual o instante em que a iteração atual é efectuada. Por fim aplica-se a atualização ao mapa, através da equação 2.6[41].

O algoritmo 6 representa o algoritmo do SOM previamente descrito.

Algorithm 6 Algoritmo de Mapa Auto-Organizado *Online*

```
1:  procedure ONLINE-SOM(inputs)
2:      map ← inicializar com n vetores aleatórios
3:      t ← 0
4:      for all i ∈ inputs do
5:          t ← t + 1
6:          distances ← ∀_k |map_k − i|
7:          bmu ← argmin(distances)
8:          map ← update_map(t, i, bmu, #inputs)
9:      end for
10: end procedure
```

2.1.2.2 Trabalho sobre Convergência

Erwin et Al. [13] fornecem uma prova da convergência para uma dimensão. No entanto eles também notam que não existe nenhuma prova de convergência para o SOM em situações com dimensões mais elevadas. Eles citam o trabalho de Lo e Bavarian [45], em que eles tentaram, sem sucesso, provar a convergência de Mapas Auto-Organizados.

Eles também mencionam que em dimensões mais elevadas, não existem estados verdadeiramente absorventes, ou seja, estados que, uma vez que o algoritmo entra, é impossível sair desse mesmo estado, o que torna impossível definir uma prova de convergência fiável para este cenário. No entanto, os autores comentam que um SOM é mais provável dirigir-se para uma configuração ordenada, do que afastar-se de tal estado.

2.1.2.3 Aplicações de Mapas Auto-Organizados

Mapas Auto-Organizados têm tido vastas aplicações desde a sua criação [37]. A seguir estão descritas algumas das aplicações que o SOM teve.

Análise Sócio-Económica Deichmann et Al. [12] utilizaram Mapas Auto-Organizados com o objetivo de analisar a convergência económica e social dos países da União Europeia desde o alargamento de 2004, analisando características como a educação, a saúde, o crescimento populacional, a riqueza, urbanização, industrialização, infraestrutura e vários índices democráticos. Temos também o trabalho de Marina Resta [63], que utilizou SOMs com o objetivo de analisar os mercados financeiros e enfatizar as relações entre diversas empresas no mercado e providenciar uma representação global da situação financeira de diversos países.

Investigação Climatérica Sheridan e Lee [66] mencionam a utilização de Mapa Auto-Organizados para classificar diferentes condições climatéricas, como por exemplo padrões climáticos, massas de ar, agrupamentos, entre outros. Temos ainda o trabalho de Tereza Cavazos [9] onde ela utiliza os mapas auto-organizados para investigar a precipitação de inverno na região das Balcãs.

Diagnóstico de Falhas Neste campo, temos por exemplo, o trabalho de Pérez et Al. [7] em que utilizaram Mapas Auto-Organizados juntamente com *Wavelets* para determinar falhas em processamento de sinais, num contexto de *model-free fault*.

Alimentação No campo da alimentação temos o trabalho de Mello et al. [54], em que utilizam SOMs juntamente com uma técnica denominada *Data Envelopment Analysis* para aglomerar os diferentes criadores de gado de algumas municipalidades do Brasil, com o objetivo de os classificar de acordo com os seus perfis de eficiência dos seus sistemas de produção.

2.1.3 Mapa Auto-Organizado Ubíquo

Enquanto que o SOM, discutido na secção 2.1.2 possui uma vasta utilidade, também possui as suas limitações, devido à sua aplicação para conjuntos de dados finitos. Os parâmetros de aprendizagem, como a taxa de aprendizagem e a função de vizinhança, apesar de garantirem a convergência, tem a necessidade de se conhecer o número total de observações *à priori*, algo que não acontece com fontes de dados ilimitadas. Existe um método que permite adaptar o SOM para fontes de dados, usando apenas as funções da taxa de aprendizagem e de vizinhança apenas nas iterações iniciais, enquanto o mapa adquire a sua ordem, mantendo esses parâmetros de aprendizagem com valores reduzidos após adquirir a ordem. No entanto, isto reduz a plasticidade do SOM, ou seja, torna-o incapaz de reagir a mudanças significativas na sequência de dados, devido à redução da magnitude das alterações feitas aos protótipos.

Tem existido trabalho de investigação para adaptar o SOM para ultrapassar essas dificuldades, como por exemplo o *parameterless-growing-SOM* [43] e o UbiSOM (*Ubiquitous Self-Organizing Map* ou Mapa Auto-Organizado Ubíquo) [69], que utilizam diversas métricas globais para adaptar os parâmetros de aprendizagem de acordo com a distribuição dos valores da fonte. Nesta secção irá ser discutido o UbiSOM, tendo por base a tese de Bruno Silva [69].

2.1.3.1 Algoritmo UbiSOM

O algoritmo UbiSOM baseia-se em duas métricas de avaliação globais, nomeadamente o erro médio de quantização e a utilidade média de neurónio, métricas calculadas sobre uma janela deslizante.

Erro Médio de Quantização O erro médio de quantização determina a tendência do processo de mapeamento para progredir na direção da distribuição subjacente á fonte de entrada. O erro de quantização local ($E'_q(t)$) é obtido a partir da equação 2.7, em que X_t é a observação atual, W_c é o protótipo selecionado para BMU e $|\Omega|$ é a maior distância entre duas entradas no espaço de entrada normalizado.

$$E'_q(t) = \frac{\|X_t - W_c\|}{|\Omega|} \tag{2.7}$$

Os valores obtidos pela equação 2.7 são depois utilizados numa janela de dimensão T (muito maior que 1, p.e. 1000) para obter $\overline{qe}(t)$, ou seja, o erro médio de quantização, descrito na equação 2.8.

$$\overline{qe}(t) = \frac{1}{T} \sum_{t}^{t-T+1} E'_q(t) \tag{2.8}$$

Utilidade Média dos Neurónios A utilidade média dos neurónios é por sua vez utilizada para detetar se existem regiões do mapa que se tornaram "inutilizadas", devido a alterações na distribuição dos dados não detetadas pela métrica anterior. Para calcular esta métrica, cada unidade do UbiSOM possuí um valor (t_k^{update}, onde k indica qual das unidades se refere) que guarda qual foi a ultima vez que aquele prototipo foi atualizado, ou seja, tenha sido a BMU ou que tenha estado na região de influência da BMU, limitado pela função de vizinhança.

A utilidade de neurónio $\lambda(t)$ é calculada de acordo com a equação 2.9, onde k identifica um neurónio e K representa o número total de neurónios.

$$\lambda(t) = \frac{\sum_{k=1}^{K} 1_{\{t - t_k^{update} \leq T\}}}{K} \tag{2.9}$$

Obtêm-se posteriormente a média dos valores de $\lambda(t)$ para obter $\overline{\lambda}(t)$, ou seja, a utilidade média dos neurónios, como formalizado na equação 2.10.

$$\overline{\lambda}(t) = \frac{1}{T} \sum_{t}^{t-T+1} \lambda(t) \tag{2.10}$$

Uma redução de $\overline{\lambda}(t)$ indica-nos que existem neurónio que não estão a ser utilizados, alertando-nos para alterações na distribuição subjacente.

Função de deriva As duas métricas são depois aplicadas em uma função de deriva que providência uma indicação do desempenho do mapa sobre uma determinada sequência de dados, sendo depois utilizada para estimar os parâmetros de aprendizagem. A função de deriva $d(t)$ encontra-se definida na equação 2.11, onde $\beta \in [0,1]$ é um parâmetro que estabelece a relação entre as duas métricas mencionadas anteriormente.

$$d(t) = \beta \overline{qe}(t) + (1-\beta)(1-\overline{\lambda}(t)) \tag{2.11}$$

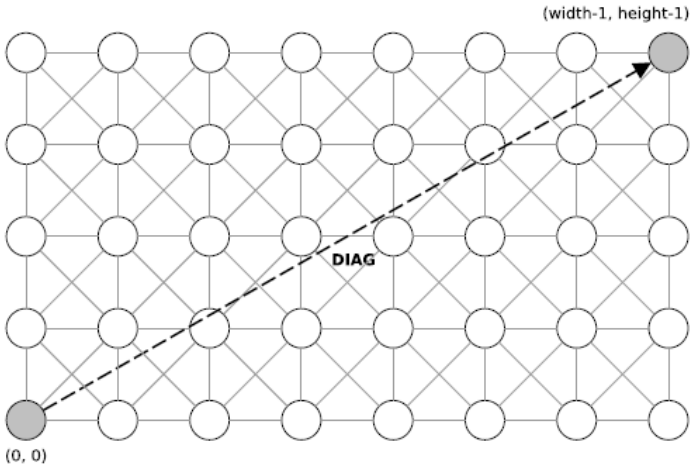


Figura 2.1: ||**DIAG**|| utilizada para normalização de $\sigma(t)$ [69]

Uma vez que $\overline{qe}(t)$ e $\overline{\lambda}(t)$ só são obtidas após T observações, o mesmo acontece com $d(t)$. Assim, antes de T, o UbiSOM não utiliza a função $d(t)$ para estimar os parâmetros de aprendizagem, mas sim parâmetros de aprendizagem monotonamente decrescentes.

Função de Vizinhança O UbiSOM utiliza um parâmetro de aprendizagem $\sigma(t)$ (definido mais à frente), ou seja, o raio da vizinhança normalizado. Esta função de vizinhança é truncada, permitindo o càlculo $\lambda(t)$.

A normalização efetuada é baseada na distância máxima entre quaisquer duas unidades na grelha do UbiSOM. Em mapas retangulares, os neurónios mais distantes são aqueles em extremidades diferentes, ou seja, posições $(0,0)$ e $(width-1, height-1)$, como ilustrado na figura 2.1. Tendo isto em consideração, podemos normalizar as distâncias dentro da grelha pela vetor euclidiano $\mathbf{DIAG} = (width-1, height-1)$, como definido na equação 2.12, limitando a vizinhança máxima que o UbiSOM pode utilizar e estabelecendo que $\sigma \in [0,1]$. Assim, o ||**DIAG**|| e o σ permitem definir o raio da vizinhança afetada.

$$\|\mathbf{DIAG}\| = \sqrt{(width-1)^2 + (height-1)^2} \tag{2.12}$$

Com isto em mente, pode-se obter a variante da função de vizinhança $h'_{ck}(t)$ utilizada pelo UbiSOM, definida na equação 2.13.

$$h'_{ck}(t) = e^{-\left(\frac{\|r_c - r_k\|}{\sigma(t)\|\mathbf{DIAG}\|}\right)^2} \tag{2.13}$$

Máquina de Estados Finitos O UbiSOM utiliza uma máquina de estados finitos de 2 estados, nomeadamente estados de ordenação e aprendizagem. O estado de ordenação permite que o mapa se ajuste à distribuição subjacente através do uso de parâmetros de aprendizagem que decrescem de forma monótona durante esse estado. É também

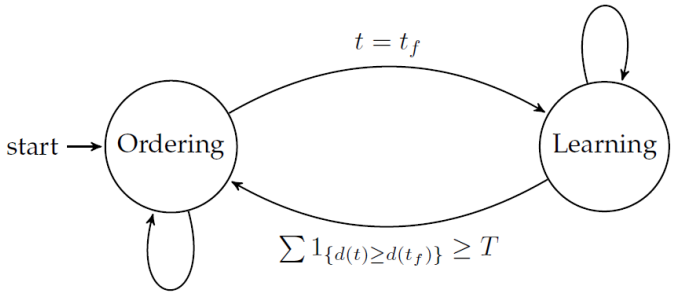


Figura 2.2: Máquina de estados finitos do UbiSOM [69]

utilizado para obter os primeiros valores das métricas de avaliação, progredindo posteriormente para o estado de aprendizagem. Isto permite ao UbiSOM preservar uma plasticidade indefinida, enquanto preserva as propriedades iniciais do SOM sobre sequências de dados não estacionárias. No entanto, caso uma mudança abrupta da distribuição subjacente seja detetada, o algoritmo regressa ao estado de ordenação.

Na figura 2.2 vemos uma representação da máquina de estados finitos do UbiSOM. Observamos que o UbiSOM inicia-se no estado de ordenação, regressando a esse estado quando existe uma mudança relevante na fonte de dados, preservando esse estado por T observações. Os parâmetros devem ser relativamente elevados, para permitir que o mapa se ajuste a partir de uma inicialização desordenada dos protótipos.

O UbiSOM necessita que se selecione os valores apropriados para η_i, η_f, σ_i e σ_f, que representam os valores iniciais e finais para a taxa de aprendizagem e o raio da vizinhança normalizado, respetivamente. O UbiSOM utiliza as equações 2.14 e 2.15 para calcular os parâmetros de aprendizagem, considerando que t_f é a ultima iteração da fase de ordenação.

$$\sigma(t) \leftarrow \sigma_i \left(\frac{\sigma_f}{\sigma_i}\right)^{\frac{t}{t_f}} \tag{2.14}$$

$$\eta(t) \leftarrow \eta_i \left(\frac{\eta_f}{\eta_i}\right)^{\frac{t}{t_f}} \tag{2.15}$$

Ao terminar a iteração t_f, o primeiro valor da função de deriva é obtido, e efetua-se a transição para a fase de aprendizagem. Nesta fase, os parâmetros de aprendizagem são estimados exclusivamente pela função de deriva $d(t)$, incrementando ou decrementando em relação ao valor de $d(t_f)$ e dos valores finais de η_f e de σ_f do estado de ordenação anterior. As equações 2.16 e 2.17 são assim utilizadas para estimar os parâmetros de aprendizagem.

$$\eta(t) = \begin{cases} \frac{\eta_f}{d(t_f)} d(t), & \text{if } d(t) < d(t_f) \\ \eta_f, & \text{c.c.} \end{cases} \tag{2.16}$$

$$\sigma(t) = \begin{cases} \frac{\sigma_f}{d(t_f)} d(t), & \text{if } d(t) < d(t_f) \\ \sigma_f, & \text{c.c.} \end{cases} \tag{2.17}$$

A consequência das equações 2.16 e 2.17 é de que se a distribuição da sequência de dados for estacionária, os parâmetros de aprendizagem acompanham a redução dos valores da função de deriva, permitindo a convergência do mapa. Por outro lado, caso existam alterações, a função de deriva aumenta, aumentando os valores dos parâmetros de aprendizagem e a elasticidade do mapa, o que permitirá que o mapa se ajuste á nova distribuição dos dados.

No entanto, caso a distribuição da sequência de dados seja de tal forma abrupta que o mapa não consegue recuperar, ou seja, não consegue convergir posteriormente, o UbiSOM regressa ao estado de ordenação. Para determinar tal circunstância, utiliza-se a equação 2.18.

$$\sum 1_{\{d(t) \geq d(t_f)\}} \geq T \tag{2.18}$$

Actualização Como podemos ver na equação 2.19, o UbiSOM utiliza a mesma regra de atualização que o *Online* SOM, adicionando um limiar minimo à função de vizinhança para o qual um determinado protótipo será atualizado.

$$W_k(t) = \begin{cases} W_k(t) + \eta(t) h'_{ck}(X_t - W_k(t)), & \text{se } h'_{ck}(t) > 0.01 \\ W_k(t), & \text{c.c.} \end{cases} \tag{2.19}$$

Formalização Tendo como base as definições anteriores de métricas de avaliação, regra de atualização, máquina de estados e métodos de estimação dos parâmetros de aprendizagem, o algoritmo do UbiSOM encontra-se formalizado no algoritmo 7.

2.2 Unidades de Processamento Gráfico

O termo "Unidade de Processamento Gráfico", ou GPU (do inglês *Graphics Processing Unit*), foi inicialmente popularizado pela NVIDIA em 1999 para descrever uma série de equipamentos optimizados para geração de imagens num computador [23]. No entanto, este termo já era utilizado pelo menos desde os anos 80 [32].

Hoje em dia, os GPUs são utilizados para muito mais do que gerar imagens num computador, tendo vastas aplicações numa área denominada GPGPU (*General-Purpose Graphics Processing Unit* ou Unidade de Processamento Gráfico de Propósito Geral) [18, 19]. Em GPGPU, os GPUs são utilizados para áreas tão distintas como:

- Computação Ciêntifica [3, 30]
- Bioinformática [49, 65]

Algorithm 7 Algoritmo de Mapa Auto-Organizado Ubíquo [69]

$inputs\_list \leftarrow$ todos os *inputs* carregados até ao momento.
$T \leftarrow$ tamanho da janela para as métricas de avaliação.
procedure UBI-SOM
    $map \leftarrow$ inicializar com n vetores aleatórios
    $t_k^{update} \leftarrow$ vetor inicializado a 0s com o ultimo *update* de cada protótipo
    $t_l^{bmu} \leftarrow$ vetor inicializado a 0s com a ultima vez que o protótipo foi selecionado como BMU.
    $t_i \leftarrow 0$
    $t_f \leftarrow t_i + T - 1$
    $t \leftarrow 0$
    $current\_state \leftarrow ordering\_state$
    for all $i \in inputs\_list$ **do**
        $distances \leftarrow \forall_k |map_k - i|$
        $bmu \leftarrow argmin(distances)$
        $t_c^{bmu} \leftarrow t$
        $\lambda(t) \leftarrow$ Calcular utilidade do protótipo com equação 2.9 e meter na fila de $\overline{\lambda}$
        $E'_q(t) \leftarrow$ Calcular erro de quantificação normalizado com equação 2.7 e meter na fila de $\overline{qe}$
        Calcular $\overline{\lambda}(t)$ e $\overline{qe}(t)$
        if $current\_state = ordering\_state$ **then**
            Calcular $\sigma(t)$ e $\eta(t)$ com equações 2.14 e 2.15.
        else
            Calcular $\sigma(t)$ e $\eta(t)$ com equações 2.17 e 2.16.
        end if
        Atualizar *map* (Equação 2.19) e t_k^{update}
        $t \leftarrow t + 1$
        if $t = t_f$ **then**
            $current\_state \leftarrow learning\_state$
        else if $\sum 1_{d(t) \geq d(t_f)} \geq T$ **then**
            $current\_state \leftarrow ordering\_state$
            $t_i \leftarrow t$
            $t_f \leftarrow t_i + T - 1$
            $\forall t_k^{update} \leftarrow t$
        end if
    end for
end procedure

- Bases de Dados [50]
- Criptografia [26–28]
- Antívirus [75]

Outra das áreas em que GPGPU é muito utilizada é a área da Aprendizagem Automática (ou *Machine Learning*), onde o poder paralelo das GPUs permite treinar Redes Neuronais com conjuntos de dados cada vez maiores[46]. Como foi visto ao longo da secção 2.1.2, os Mapas Auto-Organizáveis têm muitas oportunidades de paralelização que podem ser aproveitados com GPUs.

Nesta secção iremos discutir o funcionamento de um GPU e as suas características de operação do ponto de vista do *software*, nomeadamente do OpenCL, assim como mencionar outros modelos de programação existentes, como por exemplo, o CUDA.

2.2.1 Modelos de Programação

Na área de GPGPU há dois modelos base distintos que se destacam [60]: o CUDA, da NVIDIA, e o OpenCL, suportado pelo Khronos Group, um consórcio de empresas interessados no desenvolvimento do OpenCL e de tecnologias abertas semelhantes.

CUDA O CUDA é uma plataforma de computação paralela e um modelo de programação desenvolvido pela NVIDIA para GPGPU, com o objetivo de acelerar aplicações de computação através do poder de computação paralelo dos GPUs.

Um programa em CUDA está dividido em dois segmentos: um segmento sequencial que corre no CPU, e um segmento paralelo, onde a maioria da computação é efetuada, que corre no GPU.

Os programas CUDA são compilados com o *nvcc*, um compilador de C/C++ para LLVM desenvolvido pela NVIDIA [10].

O CUDA tem uma presença forte na área de Aprendizagem Automática, com várias ferramentas de *machine learning* suportando CUDA, como por exemplo as *frameworks Caffe*, Torch 7 e a linguagem de programação MATLAB [47].

OpenCL O OpenCL (Open Computing Language) é uma plataforma aberta para programação paralela para diversos tipos de processadores presentes em computadores pessoais, servidores, dispositivos móveis e plataformas embebidas, com o objetivo de melhorar a velocidade e a responsividade das aplicações em várias áreas de computação [58].

Enquanto que o OpenCL é bastante utilizado em aplicações cientificas e para computação visual, a sua utilização para a área de *Machine Learning* ainda é bastante rara, apenas existindo uma implementação conhecida de *Support Vector Machines* (OpenCL Support Vector Machine) [59].

Uma vez que este projeto envolve o OpenCL, irá ser elaborado o modelo de programação de OpenCL para GPUs na secção 2.2.2.

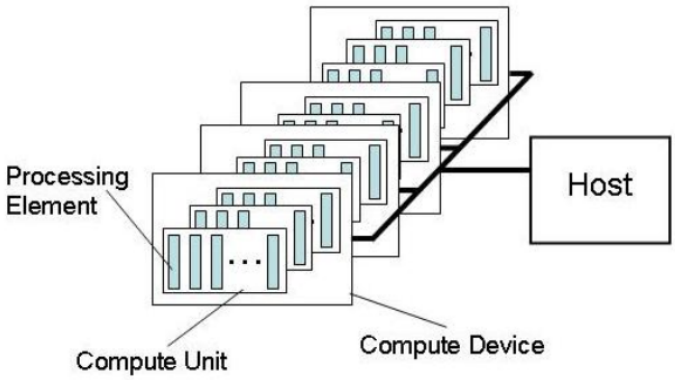


Figura 2.3: Modelo de Plataforma do OpenCL [24]

Enquanto que modelos de baixo nível como o OpenCL e o CUDA podem dar-nos acesso direto ao potencial máximo de um GPU, os detalhes de implementação muitas vezes distraem o programador da tarefa a ser efetuada. No entanto, existem modelos de mais alto nível por permitem programar para GPUs, e ao mesmo tempo, focar o raciocínio no problema em questão. Nesta secção, iremos mencionar alguns desses modelos, *frameworks* como o SkelCL, o Marrow e o *Caffe* [36, 51, 72]

2.2.2 OpenCL

O OpenCL[24] é um padrão de programação para programação paralela geral para diversas plataformas, incluindo CPUs, GPUs, entre outros, com o objetivo de permitir que os programadores consigam tirar total partido dessas plataformas.

O OpenCL é uma API para coordenar computações paralelas através de diversos processadores heterogéneos, com o objetivo de suportar diversos modelos de programação, sejam orientados aos dados ou as tarefas, que utilize uma representação intermediária portável e contida com suporte para execução paralela, com a capacidade de definir perfis para dispositivos embebidos e com interoperabilidade eficiente com outras APIs, como por exemplo, OpenGL.

2.2.2.1 Modelo da Plataforma

Como podemos ver na figura 2.3, o modelo de OpenCL consiste num dispositivo hospedeiro (por exemplo, um CPU), ligado a um ou vários dispositivos OpenCL (por exemplo, um GPU), também chamado de *Compute Device* (Dispositivo de Computação). Esse dispositivo por sua vez é divido em várias unidades de computação (CU - *Compute Unit*) que por sua vez são sub-divididas em elementos de processamento (PE - *Processing Element*), unidade onde as computações são efetuadas.

Uma aplicação de OpenCL corre num processador hospedeiro de acordo com o modelo nativo da plataforma, submetendo comandos a partir do hospedeiro para executar

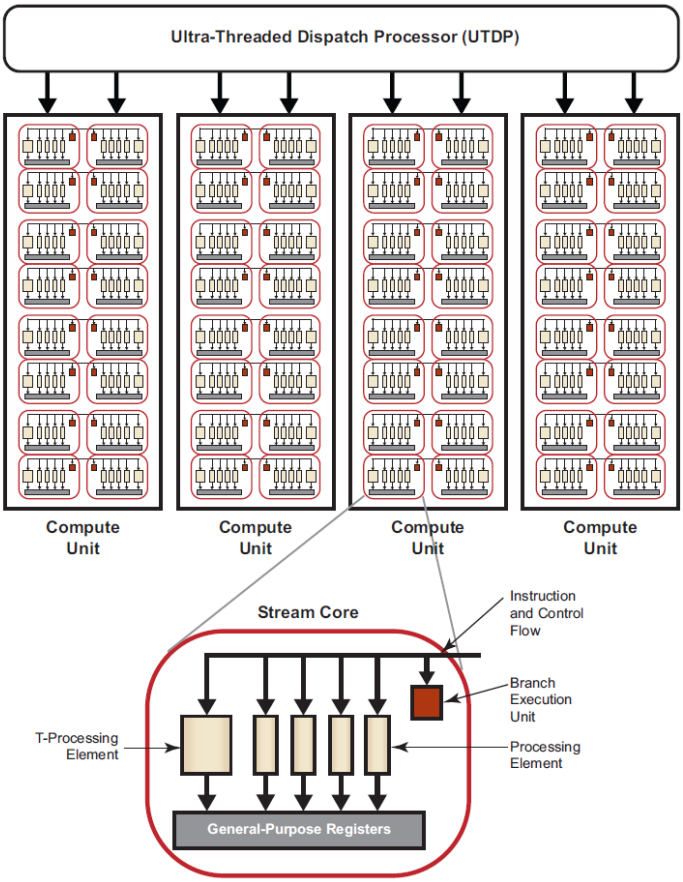


Figura 2.4: Diagrama Simplificado de um GPU (ATI/AMD) [1]

computações nos PEs dentro de um dispositivo de computação. Esses PEs dentro de cada CU executam uma única sequência de instruções como unidades SIMD ou como unidades SPMD (cada PE possui o seu próprio *Program Counter*).

Na figura 2.4 vemos um diagrama simplificado de um GPU, com termos correspondentes à nomenclatura do OpenCL.

Como podemos ver, cada GPU possui várias CUs, que por suas vez possuem múltiplos PEs. Isto permite a um GPU fazer múltiplas operações paralelas, de uma forma SIMD, sobre um determinado conjunto de dados. Como mencionado anteriormente, cada CU possui um único *Program Counter*, que mantém as PEs dentro da mesma CU sincronizadas.

2.2.2.2 Modelo de Execução

A execução de um programa OpenCL ocorre em duas partes: *kernels* que executam em dispositivos OpenCL e programa principal que executa no *host* (hospedeiro). Este programa principal define o contexto para os ***kernels*** e gere a sua execução.

O núcleo do modelo de execução do OpenCL é definido pela forma como os *kernels* executam. Quando um *kernel* é submetido para execução pelo *host*, um espaço de índices é

definido e uma instância do *kernel* executa para cada ponto desse espaço. A esta instância do *kernel* dá-se o nome de um ***work-item*** e é identificado pelo seu ponto no espaço de índices, fornecendo um ID global para o *work-item*. Cada um dos *work-items* executa o mesmo programa, mas a execução pode variar de acordo com o algoritmo e com os dados em que se opera.

Estes *work-items* são por sua vez organizados em ***work-groups***, providenciando uma decomposição com uma granularidade mais grossa do espaço de índices. Os *work-items* num determinado *work-group* executam de forma concorrente nas PEs de uma única CU.

A figura 2.5 ilustra este modelo de execução.

2.2.2.3 Modelo de Programação

O OpenCL suporta dois tipos distintos de modelos de programação: **paralelização de dados** e **paralelização de tarefas**, assim como existe suporte para modelos híbridos desses dois formatos. No entanto, o modelo principal do OpenCL é a paralelização de dados, e por isso, vamos focar-nos nesse modelo.

O modelo de programação de paralelismo de dados define uma computação como sendo uma sequência de instruções aplicadas a múltiplos elementos em memória. O OpenCL permite assim definir um mapeamento dos dados para os *work-items*, como exemplificado na figura 2.5.

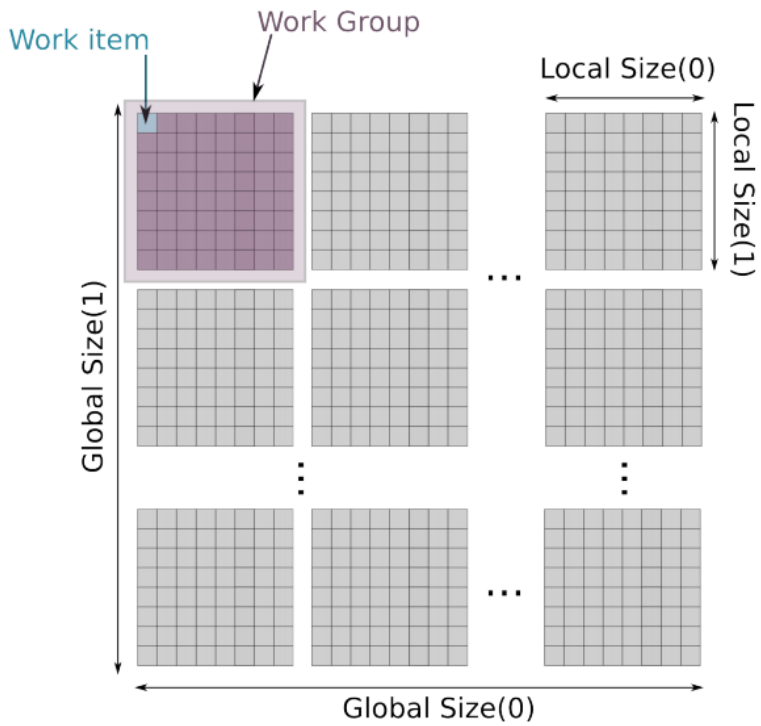


Figura 2.5: Paralelismo de Dados em OpenCL [8]

Assim, cada *Work Group* pode ser atribuído a uma *Compute Unit* e por sua vez, cada

Work Item poderá ser atribuído a um *Processing Element*.

No entanto, é importante notar que o OpenCL providencia um modelo hierárquico, com duas maneiras distintas de especificar a sub-divisão. Num modelo explícito, o programador define o número total de *work-items* a executar em paralelo e como é que esses *work-items* são divididos entre os diferentes *work-groups*. No modelo implícito, o programador especifica apenas o número total de *work-items* a executar em paralelo, sendo a sua divisão gerida pela implementação do OpenCL em questão.

Por outro lado, temos o modelo de programação de paralelismo de tarefas. Este modelo é definido por uma única instância de um *kernel* que é completamente independente do conceito de *work-items*, sendo o equivalente (em termos de *data-paralelism*) a executar um único *kernel* numa CU com um único *work-item*. Neste modelo, o programador pode exprimir paralelismo por utilizar tipos de dados vetoriais implementados pelo dispositivo, meter várias tarefas em fila ou por colocar em fila *kernels* nativos desenvolvidos utilizando um modelo de programação ortogonal ao OpenCL.

2.2.3 Marrow Expressions

Os GPUs são uma fonte de capacidade de processamento disponível em qualquer dispositivo comercial. No entanto, para aproveitar esse poder de processamento, é necessário programar especificamente para esse dispositivo, utilizando plataformas como o CUDA e o OpenCL, ou outros modelos de mais alto nível como o SkelCL ou o Caffe.

O objetivo das Marrow Expressions [15, 16, 61, 70] é permitir que o programador se concentre mais no aspeto lógico do desenvolvimento de sistemas e menos nos detalhes de implementação, detalhes que muitas vezes são específicos a determinados dispositivos.

Marrow Expressions são construídas por cima do *Marrow Algorithmic Skeleton Framework* [51], uma *framework* de esqueletos algorítmicos para GPGPU. Também possui várias otimizações para unir a simplicidade no desenvolvimento com ganhos de performance em vários cenários.

A biblioteca Marrow Expressions fornece um conjunto de operações básicas e de estruturas de dados que com o uso de templates de expressões, geram código OpenCL automaticamente assim como orquestram a sua execução no GPU, simplificando assim o trabalho do programador.

Estruturas de Dados Marrow Expressions providência algumas estruturas de dados para serem utilizadas no desenvolvimento de aplicações heterogéneas, nomeadamente, *arrays*, vetores e matrizes. Estas estruturas de dados mimicam o comportamento normal das estruturas correspondentes da biblioteca *standard* do C++, porém, possuem lógica adicional que permite serem enviadas para dentro de um GPU.

- **Escalar** Representa um único valor.

- **Array** Estrutura de dados de tamanho estático, declarada em tempo de compilação. Logicamente equivalente ao std::array.
- **Vector** Estrutura de dados de tamanho dinâmico, alocadas em tempo de execução. Logicamente equivalente ao std::vector.
- **Matriz** Estrutura de dados multi-dimensional, cujo tamanho pode ser declarado em tempo de compilação (tamanho estático) ou em tempo de execução (tamanho dinâmico).

Operações Também são fornecidas várias operações, que podem ser combinadas para gerarem *kernels* OpenCL. Isto funciona porque ao declarar cada uma dessas operações, uma abstract syntax tree (AST, em português: Árvore Abstrata de Sintaxe) é gerada e a partir daí, o código OpenCL é gerado automaticamente. Exemplos dessas operações são:

- **Atribuições** Atribuem valores a uma estrutura de dados.
- **Operações Binárias** Aplica uma operação binária a duas estruturas de dados, como por exemplo uma adição.
- **Operações unárias** Aplica uma operação unária a uma estrutura de dados, por exemplo, um incremento.

Funções Definindo as estruturas de dados e as operações base, como adição, subtração, divisão e multiplicação, isto permite-nos definir funções e algoritmos cada vez mais complicados, que podem ser aplicadas com as estruturas de dados já mencionadas. Um exemplo desses funções que já foram implementadas na biblioteca são funções de cálculo de distância euclidiana e de produto interno que já foram implementadas no Marrow.

Composição de funções O *backend* do Marrow Expressions, a Marrow Algorithmic Skeleton Framework, tem a capacidade de efetuar aninhamento de esqueletos, definindo uma AST (*Abstract Syntax Tree*, ou Árvore de Sintaxe Abstrata) que permite compor o que seriam vários programas OpenCL [16, 51]. Essa composição é feita por utilizar o tipo do C++ auto (como demonstrado na listagem 2.1), ou por utilizar o operador «. Isto permite compor execuções como por exemplo na listagem 2.1, onde uma função *sqr_distance*, que devolve o quadrado da distância euclidiana, aceitando dois *arrays* como argumentos, está definida. Nesta função, começamos por definir a AST da imagem 2.6:

- Na linha 4 cria-se a sub-árvore *subs* com uma operações binária de subtração entre os *arrays left* e *right*, á variável *subs*.
- Na linha 5 cria-se outra sub-árvore, denominada *powers*, com uma operação binária de multiplicação entre duas instâncias da sub-árvore *subs*.

Listagem 2.1: Exemplo de composição de operações utilizando a *keyword* auto

```
1 template <typename T>
2 T sqr_distance(array<T>& left, array<T>& right)
3 {
4   auto subs = left - right;
5   auto powers = subs * subs;
6   scalar<T> result = marrow::reduce<plus>()(powers);
7   return result.value();
8 }
```

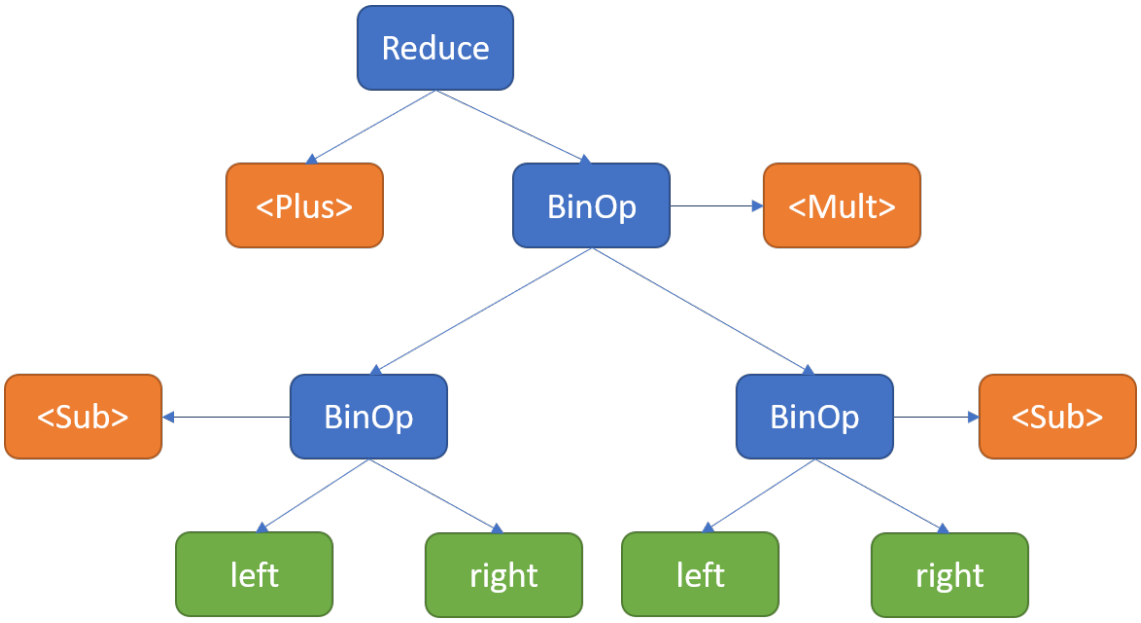


Figura 2.6: AST gerada por listagem 2.1. Blocos azuis representam nós de operações, blocos laranja representam a função associada á operação e blocos verdes representam tipos de dados.

- Na linha 6, cria-se a raiz do AST, com uma redução com a operação de soma a partir do resultado da sub-árvore *powers*.
- Na linha 7, manda-se executar o código OpenCL gerado pela AST e retorna-se o valor resultante.

2.3 Mapas Auto-Organizados em Unidades de Processamento Gráfico

Devido à popularidade do Mapa Auto-Organizado e à sua vasta aplicação, é natural procurar maneiras de acelerar o processamento dos SOMs, principalmente para grandes mapas. Uma dessas formas foi adaptar o uso de GPUs e do seu poder de processamento paralelizado para acelerar algumas etapas do processamento do SOM [52, 77]. Esta adaptação

tem permitido o uso de conjuntos de dados cada vez maiores, facilitando o seu uso para *data mining* [53] e para aplicações médicas [11].

Alguns procuraram acelerar o SOM por implementarem a procura da BMU em GPUs, utilizando técnicas de *MapReduce* [52, 77]. Surgiram também bibliotecas como por exemplo o *Somoclu* [78], com o objetivo de fornecerem uma implementação confiável do SOM para GPUs. Outros ainda procuraram escalar para ainda mais além, procurando adaptar o SOM para *clusters* de GPUs [53]. Wittek e Darányi [77] adaptaram uma variante do SOM, o Batch-SOM, para executar sob o modelo de *MapReduce*. Wittek e Darányi também mencionam que ao calcular a BMU, pode-se utilizar o quadrado da distância euclidiana em vez da distância euclidiana, uma vez que o cálculo da raiz quadrada é uma operação bastante cara num GPU.

Methew e Joy ainda mencionam outras otimizações para melhorar a performance do processamento em GPU, como evitar acessos à memoria global não coalescidos (ou seja, evitar situações em que as *threads* acedem a memória global em tempos distintos), e através do uso de memórias partilhadas e de textura do GPU, utilizando *texture binding* para fazer *caching* dos *buffers* utilizados nas computações. Eles também mencionam a necessidade de procurar lotar a capacidade de processamento de cada unidade de computação para melhor performance [52].

Os trabalhos mencionados anteriormente [52, 55, 77] chegaram todos á conclusão de que a implementação da técnica de *MapReduce* em GPUs beneficiou bastante a performance do SOM, chegando a ter um aumento de performance de até 84 vezes numa determinada situação [52].

2.4 Considerações Finais

Neste capitulo foram analisados alguns algoritmos de Análise Exploratória Avançada de Dados, como o K-Médias, o SOM e o UbiSOM, podendo compreender assim as relações entre cada um destes algoritmos e o seu funcionamento. Observou-se que em certa medida, o SOM é essencialmente um K-Médias com um valor de K elevado e com um conceito de mapa, possuindo assim características que não são aplicadas ao K-Médias. Por outro lado, o SOM é um algoritmo que evolui com a introdução de novas observações, enquanto que o K-Médias possuí um processo iterativo que depende de todas as observações estarem disponíveis no inicio. Ao aproveitar essa capacidade adaptativa do SOM, o UbiSOM estende-o para ser capaz de lidar com fontes de dados infinitas, ao adicionar parâmetros e métricas que fornecem uma maior elasticidade ao mapa, de acordo com a distribuição atual da fonte de dados.

Outro foco foi a utilização de GPUs para acelerar o processamento de dados em computadores. Devido ao número elevado de elementos de processamento, é possível utilizar o GPU para acelerar de forma eficiente trabalhos que são muito paralelizáveis. Foi também analisado as diferentes ferramentas existentes para desenvolver programas que executam

em GPUs, com um maior foco no OpenCL e no Marrow. E, tendo em base o modelo do OpenCL, observou-se como é que um GPU executa esses mesmo programas em paralelo.

Também se analisou o trabalho desenvolvido em implementar o SOM em GPUs, permitindo assim analisar as diferentes otimizações que diferentes autores implementaram ao desenvolverem as suas implementações. Otimizações essas que são úteis para o UbiSOM, uma vez que ele continua a partilhar muitas partes da sua execução com o algoritmo Online-SOM original.

A seguir, vai-se detalhar o trabalho executado ao implementar o UbiSOM em GPU, utilizando a framework Marrow.

Capítulo 3

Algoritmo UbiSOM em GPU

3.1 Arquitetura

Neste capítulo será elaborada a arquitetura da implementação em questão. Vamos discutir o modelo da infraestrutura que rodeia o algoritmo UbiSOM e que é fundamental para que o algoritmo possa receber as observações utilizadas no seu treino e os comandos necessários para a sua utilidade. Para além disso, irá ser discutido o modelo de memória usado no algoritmo, nomeadamente quais são as estruturas de dados que têm uma condição permanente ou temporária na memória do GPU e as trocas de dados entre o CPU e o GPU.

3.1.1 Modelo do Sistema

Podemos observar na figura 3.1 o modelo adotado para esta implementação. Na figura 3.1 é possível observar um servidor, que recebe de forma contínua conjuntos de dados discretizados de tamanho arbitrário a partir de uma fonte de dados, devido a uma *pipeline* para tratamento do fluxo de dados, posteriormente elaborada na secção 3.1.1.1.

Existe ainda uma relação de acordo com o modelo cliente-servidor com uma aplicação visualizadora. Essa aplicação visualizador pode fazer vários pedidos ao servidor, como por exemplo, pedir o estado atual do mapa, pedir as BMUs para as últimas n épocas do treino e ainda pedir para obter a BMU para uma determinada observação. Esses pedidos são processados através de uma *pipeline* de tratamento de comandos, que será elaborada posteriormente na secção 3.1.1.2.

3.1.1.1 *Pipeline* de Dados

A figura 3.2 ilustra de uma perspetiva de alto nível a forma como o servidor processa os *batches* discretizados que vão chegando. No estágio (1) essas observações são recebidas a

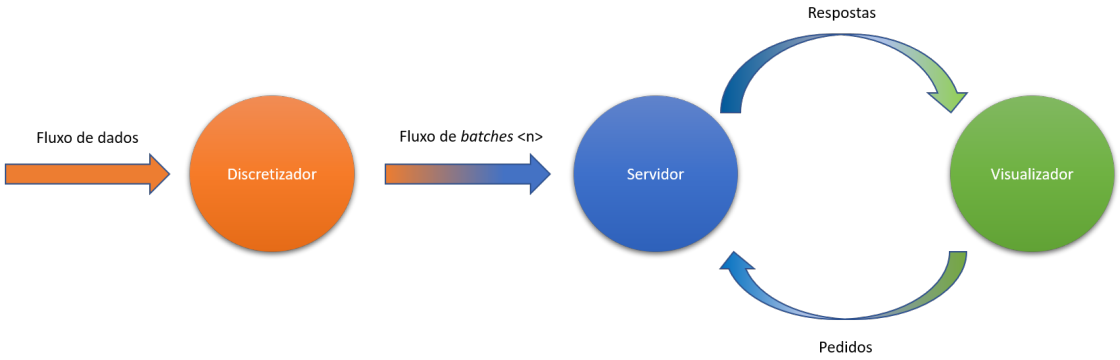


Figura 3.1: Modelo do Sistema

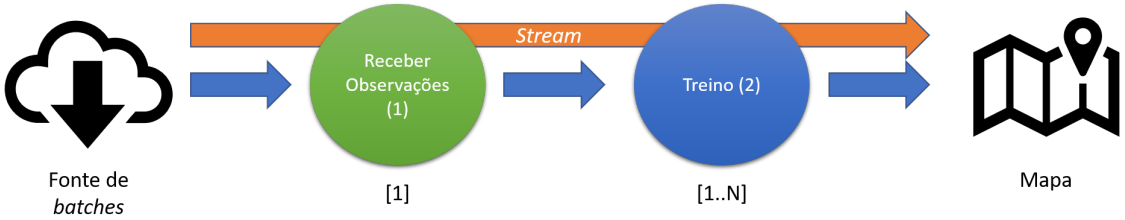


Figura 3.2: *Pipeline* de processamento da fonte de dados.

partir da fonte de *batches*, recebidos a partir de um socket TCP. Esses *batches* são depois encaminhados para o estágio seguinte da *pipeline*, o estágio (2), onde o algoritmo do UbiSOM é executado, e os resultados escritos no mapa.

A figura 3.3 detalha o que ocorre dentro do estágio de treino (2) da *pipeline*, Cada observação no *batch* que está a ser processado passa por uma *pipeline* que efetivamente é executada no CPU. Essas etapas consistem (1) no cálculo das distâncias entre cada unidade e a observação, (2) na selecção da BMU e (3) na atualização do mapa. Cada uma dessas etapas, de forma simultânea, comunica com o GPU para executar as partes do algoritmo que precisam do paralelismo de dados fornecido pelo GPU, providênciando também paralelismo de tarefas. Assim que todas as observações do *batch* atual tiverem sido processadas, executa-se o (4) cálculo das métricas do UbiSOM, o qual também aproveita os recursos do GPU quando necessário.

É possível obter um nível de paralelismo ainda maior ao aumentar o número de *pipelines* a serem executadas em paralelo. A figura 3.4 ilustra essa capacidade: ao receber os *batches* do discretizador, o estágio 1 da *pipeline* do 1º nível pode distribuir os *batches* pelas diferentes *pipelines* de treino (2) à medida que estas vão ficando disponíveis para novos trabalhos.

Para além da versão com *pipeline* dentro do estágio de Treino, foi implementada uma versão sequencial do algoritmo. Para além do paralelismo de dados no GPU, é possível ter várias instâncias do algoritmo de treino a correr em paralelo, de forma semelhante à existência de múltiplas *pipelines*.

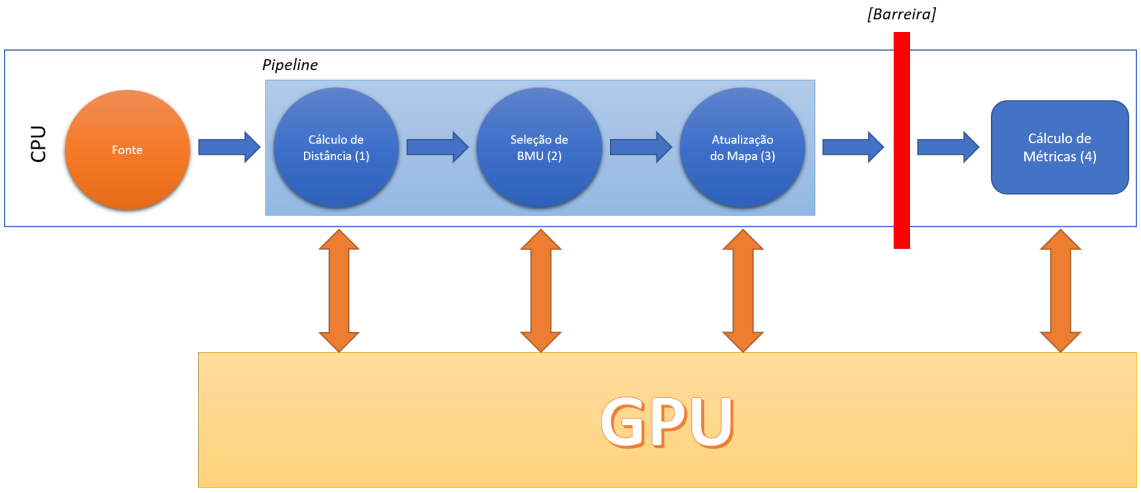


Figura 3.3: Detalhe do estágio 2 da *pipeline* na figura 3.2.

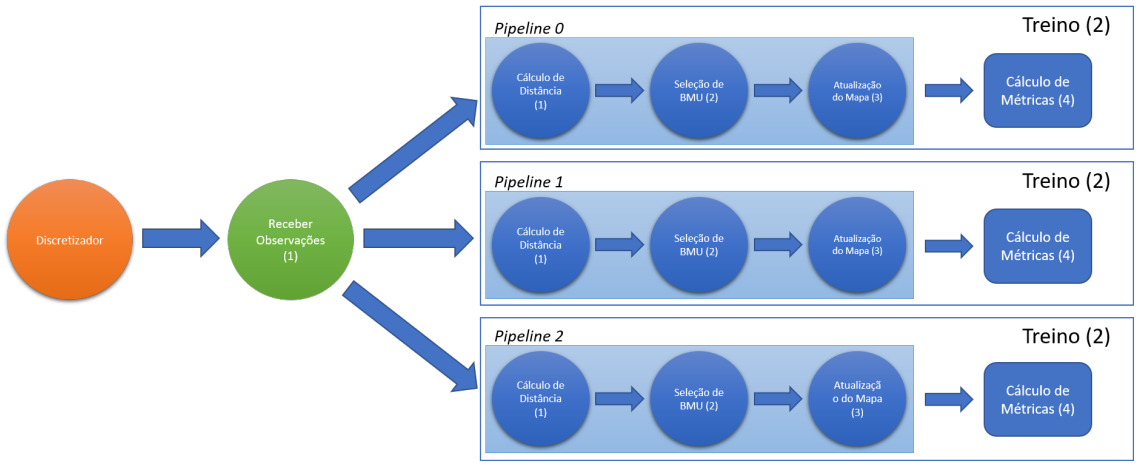


Figura 3.4: Execução de múltiplas *pipelines*.

3.1.1.2 *Pipeline* de Comandos

A figura 3.5 ilustra o funcionamento do servidor em relação ao processamento de comandos oriundo a partir das aplicações visualizadoras. Esses comandos comunicados através de um canal TCP atravessam três estágios: (1) Receber pedido (ou comando), (2) Processar o pedido, (3) Enviar resposta. O estágio 1 é responsável por receber o pedido e interpretar a mensagem (formato elaborado na secção 3.4), passando-o para o estágio seguinte. O estágio 2 faz o processamento do pedido, interagindo com o estado do mapa em memória do GPU. O estágio 3 é responsável por responder ao pedido com a resposta apropriada.

3.1.2 Modelo de Memória

Na figura 3.6, vemos a alocação de memória e o fluxo de dados existente durante a execução dos vários estágios da *pipeline*. Informações como o mapa do UbiSOM, o cálculo das distâncias entre unidades e o *array* com a última atualização, são carregadas na fase

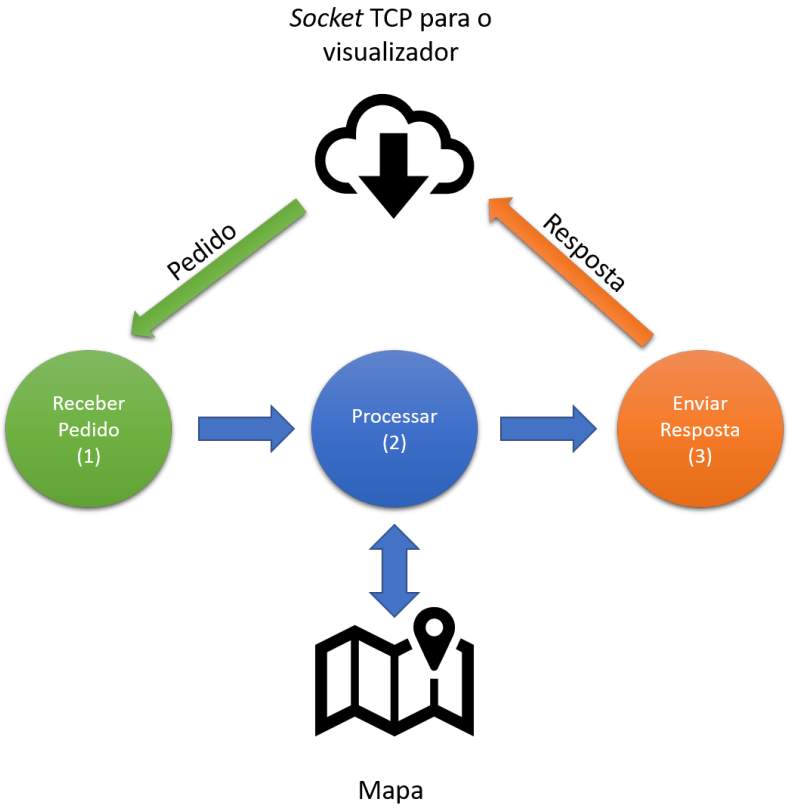


Figura 3.5: *Pipeline* de processamento de comandos.

(0) de preparação, sendo esse estágio 0 um estágio "lógico"que representa quaisquer operações efetuadas antes da execução da *pipeline*. Essas informações são armazenadas de forma persistente no GPU, ao longo de toda a execução. As alocações de memória no GPU temporárias e as transferências de memória respectivas para cada estágio são elaboradas na secção 3.2.

Depois da execução completa da *pipeline*, existe o cálculo das métricas do UbiSOM, para analisar o estado atual do algoritmo. A figura 3.7 ilustra o modelo de alto nível das transferências de memória nesse processo. É de notar que o nó Treino refere-se à *pipeline* descrita anteriormente, na secção 3.1.1.1.

Para a fase de Treino, é possível observar que o mapa UbiSOM e o mapeamento das últimas atualizações são as informações persistentes necessárias. As transferências para cada um dos cálculos das métricas são elaboradas na secção 3.2.4.

3.1.2.1 Casos especiais

Existem transferências não descritas anteriormente que podem ocorrer em ocasiões pontuais. Por exemplo, a primeira vez que estruturas de dados como o mapa (de dimensão $feature\_size * lines * columns$, valores descritos na tabela 3.1), as distâncias entre as unidades (de dimensão $lines * lines * columns * columns$), e a última atualização das unidades

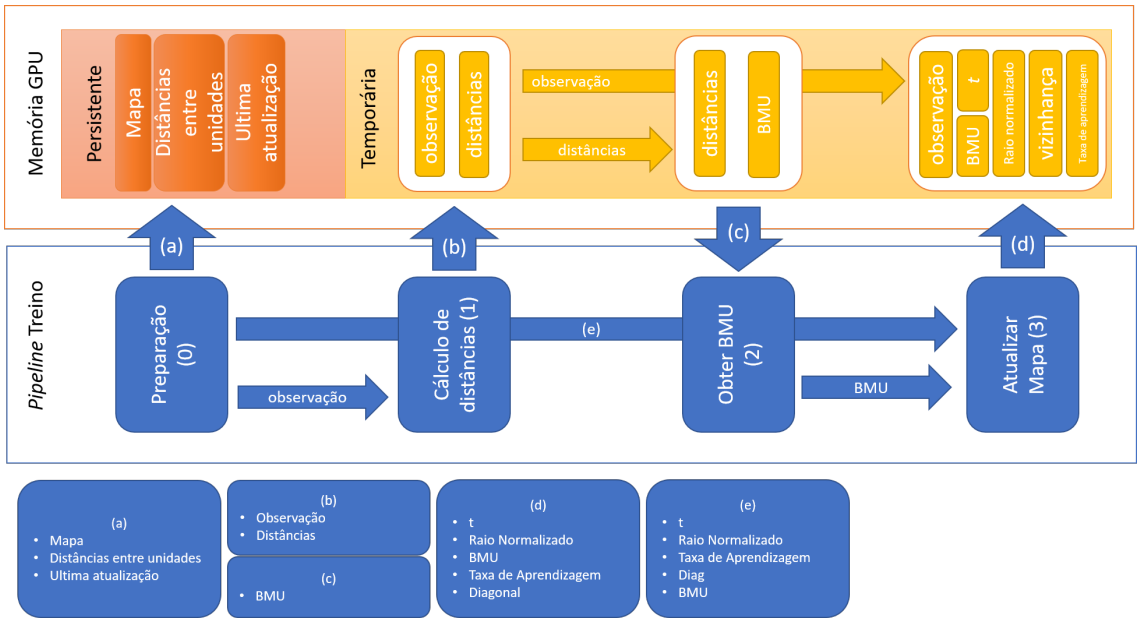


Figura 3.6: Transferências de memória durante a execução da *pipeline*.

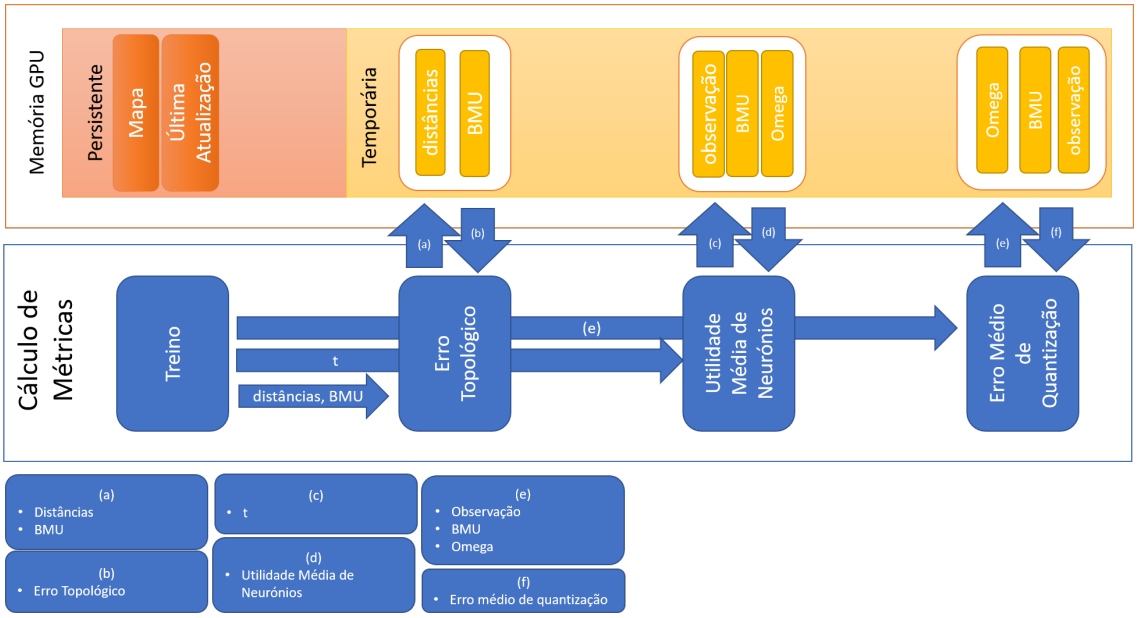


Figura 3.7: Modelo de alto nível das transferências de memória ao processar as métricas do UbiSOM.

Listagem 3.1: Método Marrow que efetua o Cálculo da Distância.

```
1 marrow::array<type, size_dim1>;
2 marrow::matrix<type, size_dim1, size_dim2>;
```

Tabela 3.1: Parâmetros do *template* da classe do algoritmo UbiSOM

Nome do Parâmetro	Tipo	Descrição
feature_size	int	O número de caracteristicas de cada unidade e observação.
lines	int	Número de linhas do mapa.
columns	int	Número de colunas do mapa.
window_size	int	Dimensão da janela utilizada para calcular as métricas do UbiSOM.

(de dimensão $lines * columns$), entre outras, são invocadas, essas terão que ser transferidas do CPU pelo GPU pela primeira vez, uma vez que tais informações não estão lá presentes.

Outros caso especial é quando um comando enviado por um visualizador precisa de acesso ao mapa. Com comandos como por exemplo, requisitar informações do mapa (elaborado na secção 3.4.1), estruturas como o mapa tem que ser transferido do GPU para o CPU para poder popular a resposta a esse comando. Operações como por exemplo, para definir um estado novo do mapa (secção 3.4.1) implicam transmitir o estado novo do CPU para o GPU quando necessário.

3.2 Implementação do Algoritmo UbiSOM

A implementação do UbiSOM, focou-se em 3 componentes principais. Cada um desses componentes corresponde às partes paralelizáveis do algoritmo, nomeadamente: cálculo da distância euclidiana entre as unidades, seleção da BMU e atualização do mapa. Para além disso outros métodos implementam o cálculo das métricas do UbiSOM: o Erro Topológico, o Erro Médio de Quantização e a Utilidade Média de Neurónios.

É ainda de notar que o algoritmo foi implementado utilizando a funcionalidade de *template* do C++. Os parâmetros do *template* estão descritos na tabela 3.1. Estes parametros são definidos pelo programador, e tendo um tipo *unsigned int* de 32bits, podem variar entre 1 e 2^{32}.

As estruturas de dados Marrow utilizadas mais frequentemente neste algoritmo são o array, e a matriz, com as parametrizações de *template* indicadas na listagem 3.1, onde *type* é o tipo do conteúdo, $size\_dim1$ é o tamanho da primeira dimensão e $size\_dim2$ é o tamanho da segunda dimensão.

A tabela 3.2 indica o nome de algumas colecções utilizadas ao longo do algoritmo. Os campos vazios na 2ª dimensão ocorrem porque as *arrays* do Marrow apenas têm uma dimensão.

Tabela 3.2: Colecções utilizadas ao longo do algoritmo.

Nome da Colecção	Tipo	Tipo dos dados	Dimensão 1	Dimensão 2	Descrição
som_map	matrix	float	lines * columns	feature_size	Mapa do UbiSOM. Cada valor do mapa é armazenado como um número real de virgula flutuante de precisão simples.
observation	array	float	feature_size		Coleção com a observação a ser processada, expressa como um conjunto de valores reais com virgula flutuante de precisão simples.
distances	array	float	lines * columns		Coleção com as distâncias entre a observação e cada uma das unidades do mapa. Distâncias expressas como valores reais com virgula flutuante de precisão simples.
t_k_update	array	int	lines * columns		Coleção com a ultima atualização de cada unidade, expressa como um valor inteiro de precisão simples.
unit_distances	matrix	float	lines * columns	lines * columns	Matriz com mapeamento de distâncias entre cada par de unidades do mapa, expressa como um valor real de virgula flutuante de precisão simples.
topological_error_queue	array	float	window_size		Fila de erros topológicos. Cada valor de erro topológico é armazenado como um número real de virgula flutuante de precisão simples.
quantization_error_queue	array	float	window_size		Fila de erros de quantização. Cada valor de erro topológico é armazenado como um número real de virgula flutuante de precisão simples.
neuron_utility_queue	array	float	window_size		Fila de utilidade de neurónio. Cada valor de erro topológico é armazenada como um número real de virgula flutuante de precisão simples.

3.2.1 Cálculo do Quadrado da Distância euclidiana

O método *map_distances*, descrito na listagem 3.2, implementa o 1º método dos 3 métodos principais implementados. Esse método é responsável por calcular o quadrado da distância euclidiana entre uma determinada observação e cada unidade no mapa *som_map*, calculando assim o segmento de $\|x - m_i\|$ da equação 2.4. O resultado é armazenado no *array Marrow distances*.

O Marrow cria um *kernel* [1] com os conteúdos das linhas 4 e 5 (Apêndice A, Listagem A.1), executando-o quando os seus resultados são armazenados na matriz *power*. Essa

[1] O Marrow faz a divisão do algoritmo em vários *kernels* (como descrito na secção 2.2.3) para simplificar os atos de sincronização quando necessário. Por exemplo, depois de várias operações algébricas entre colecções de dados, onde não é necessário haver nenhuma sincronização entre cada uma das operações, pode existir uma operação de redução para o qual é necessário ter todos os resultados das operações algébricas disponíveis. Para além disso pode ser necessário um número de Elementos de Processamento (ver secção 2.2.2.1) diferentes. A maneira mais simplificada de obter isto, é dividir a operação em *kernels* distintos, passando apenas um apontador de um *buffer* de memória intermédio entre a execução de cada *kernel*.

Listagem 3.2: Método Marrow que efetua o Cálculo da Distância.

```
1 void map_distances(array<float, feature_size>& observation,
2    array<float, lines * columns>& distances)
3 {
4     auto obs_map_diff = som_map - observation;
5     auto power = obs_map_diff * obs_map_diff
6     distances = reduce<plus<float>>() (power);
7 }
```

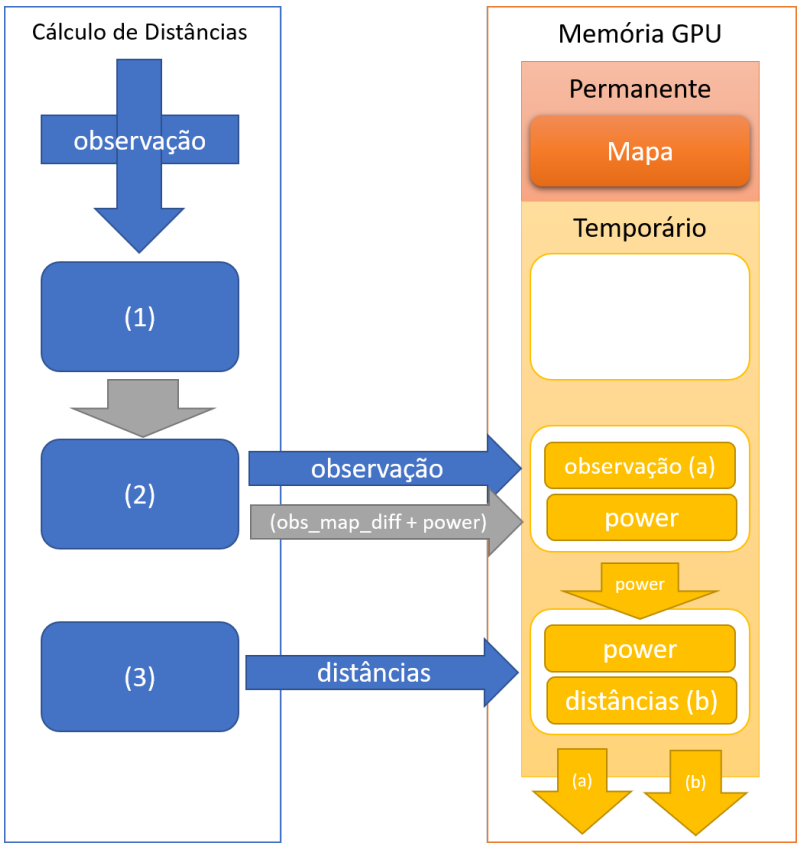


Figura 3.8: Transferências de memória durante a execução do cálculo de distâncias.

matriz é depois depois reduzida para um *array* que é guardado no campo *output* (*Kernel* descrito na secção 3.3.1).

A figura 3.8 demonstra as transferências de memória ocorridas durante a execução deste método. A primeira operação, sendo uma composição das operações (1) e (2), coloca a observação no GPU e aloca o *buffer power* em GPU. Esse *buffer* é depois utilizado na ultima operação, que após alocar o *buffer* distâncias, coloca nesse *buffer* os resutlados do cálculo de distância. Os *buffers* em GPU com a observação e com as distâncias são passados para a operação seguinte.

Listagem 3.3: Método Marrow obtém a BMU.

```
1 int get_bmu(array<float, lines * columns>& distances)
2 {
3     coordinate<1> bmu_reduce = reduce<func::min<float>,
4        reduction_target::indexes>() (distances);
5     return bmu_reduce[0];
6 }
```

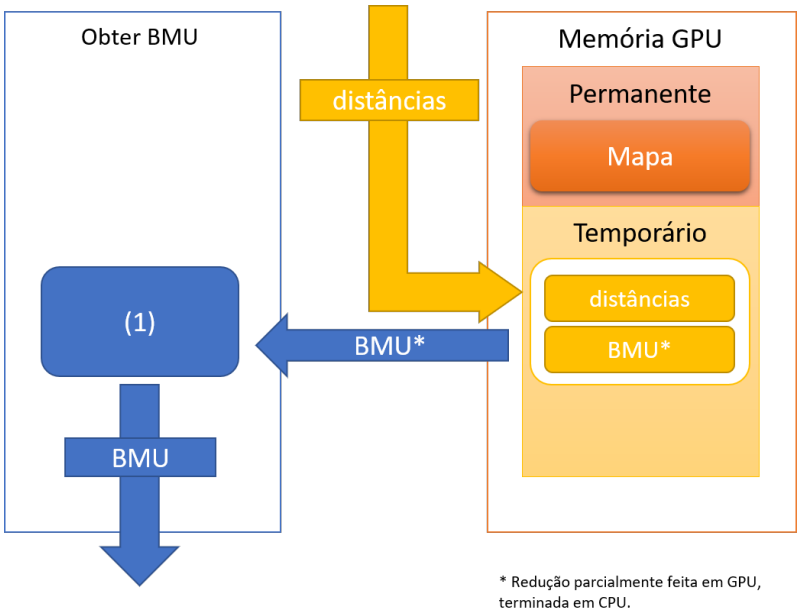


Figura 3.9: Transferências de memória durante a execução do cálculo de distâncias.

3.2.2 Seleção da BMU a partir das distâncias

O 2º método, descrito na listagem 3.3, faz a selecção da BMU a partir do mapeamento de distâncias efetuado no 1º método, por computar o segmento de minimização na equação 2.4. Essas distâncias são passadas pelo parâmetro *distances*. O 1º parâmetro de *template* da operação de redução indica que se vai aplicar uma função de minimização a *floats*. O segundo parâmetro, indica que em vez de se devolver o valor mínimo, vai ser retornado o índice corresponde a esse valor.

O Marrow executa um *kernel* OpenCL para fazer a maior parte da redução no GPU. No entanto, ao chegar a um instante em que se perde a vantagem de paralelismo do GPU, retorna os valores restantes para o CPU onde termina a redução. Esse *kernel* está descrito na listagem A.2 no apêndice A.

A figura 3.9 ilustra as transferências de memória durante esta operação. Como visível no diagrama, o *buffer* de distâncias já está presente no GPU, oriundo da tarefa anterior. Esse *buffer* é parcialmente reduzido no GPU, no entanto, ao chegar a um número pré-determinado de elementos, esses elementos são transferidos para o CPU, onde a redução é terminada. O resultado dessa redução, a BMU é encaminhada para a tarefa seguinte.

3.2.3 Atualização do Mapa

O 3º método, listado na listagem 3.4, efetua a atualização do mapa (segundo a equação 2.19). O método *update_map* aceita como argumentos o índice da unidade selecionada como BMU, os valores atuais da taxa de aprendizagem e do raio de vizinhança normalizado e a observação atual. Este método começa por fazer o cálculo da função de vizinhança (Equação 2.13) para cada uma das unidades do mapa em relação à BMU. Após obter os resultados da função de vizinhança (guardados em $neighborhood_2$), esses são utilizados para calcular, após multiplicação com a taxa de aprendizagem, a magnitude das modificações para cada unidade (guardadas na variável $modifier$). Essa magnitude é aplicada à diferença entre a observação e cada unidade do mapa, sendo o seu resultado guardado em $diff$, na linha 15. Por fim, na linha 16, soma-se esse resultado ao mapa *som_map*. É de notar também que se aproveita o método *update_map* para atualizar o vetor *t_k_update* com o instante em que cada unidade foi atualizada pela ultima vez.

Nas linhas 5 a 8, o Marrow constrói um *kernel* com o objetivo de processar a função de vizinhança, que é executado ao preencher o *array neighborhood* (*kernel* na listagem A.3, no apêndice A). Nas linhas 10 a 13, um segundo *kernel* é gerado, para calcular a magnitude das modificações que cada unidade vai sofrer (Listagem A.4 no apêndice A). Esse *kernel* é gerado e executado ao preencher o *array* $modifier$. Um terceiro *kernel* é gerado nas linhas 15 e 16, onde a atualização do mapa é realmente afetada, envolvendo a observação atual, o estado do mapa atual, e uma versão transposta do *array* $modifier$, que utiliza um *kernel* próprio para operar com esse *array* transposto. (Listagem A.5, apêndice A). Um quarto e último *kernel* é gerado na linha 18, onde se faz a atualização do *array t_k_update* (Listagem A.6, apêndice A).

A figura 3.10 demonstra as tranferências de memória na execução deste segmento do algoritmo. A observação já está presente no GPU. No entanto, ao longo da execução, o raio normalizado, a diagonal do mapa, a BMU, a taxa de aprendizagem e a iteração atual t são enviados do CPU para o GPU. É importante perceber que este método interage bastante com informações que estão permanentemente no GPU, como o mapa do UbiSOM, o mapeamento da distância entre as unidades do mapa e o *buffer* com os instantes em que cada unidade foi atualizada pela última vez.

3.2.4 Implementação das Métricas UbiSOM

Para além destes *kernels* principais, de onde se espera a maior parte dos ganhos de desempenho, vários *kernels* secundários foram implementados para acelerar algumas operações de suporte ao algoritmo UbiSOM, como, por exemplo, no cálculo do erro topológico, do erro médio de quantização e da utilidade média de neurónio.

Listagem 3.4: Método Marrow atualiza o mapa.

```
void update_map(int bmu, float normalized_radius, float learning_rate,
   array<float, feature_size>& observation)
{

    auto d = unit_distances[bmu];
    auto upper = marrow::pow(unit_distances[bmu] /
      (normalized_radius * diag), 2.0) * -1.0;
    array<float, lines * columns> neighborhood = marrow::exp(upper);

    auto neighborhood_2 = conditional(neighborhood > 0.01,
      neighborhood, 0.0);
    array<float, lines * columns> modifier = neighborhood_2
        * learning_rate;

    auto diff = (observation - som_map) * modifier.as_column();
    som_map = som_map + diff;

    t_k_update = conditional(neighborhood > 0.0, t, t_k_update);

}
```

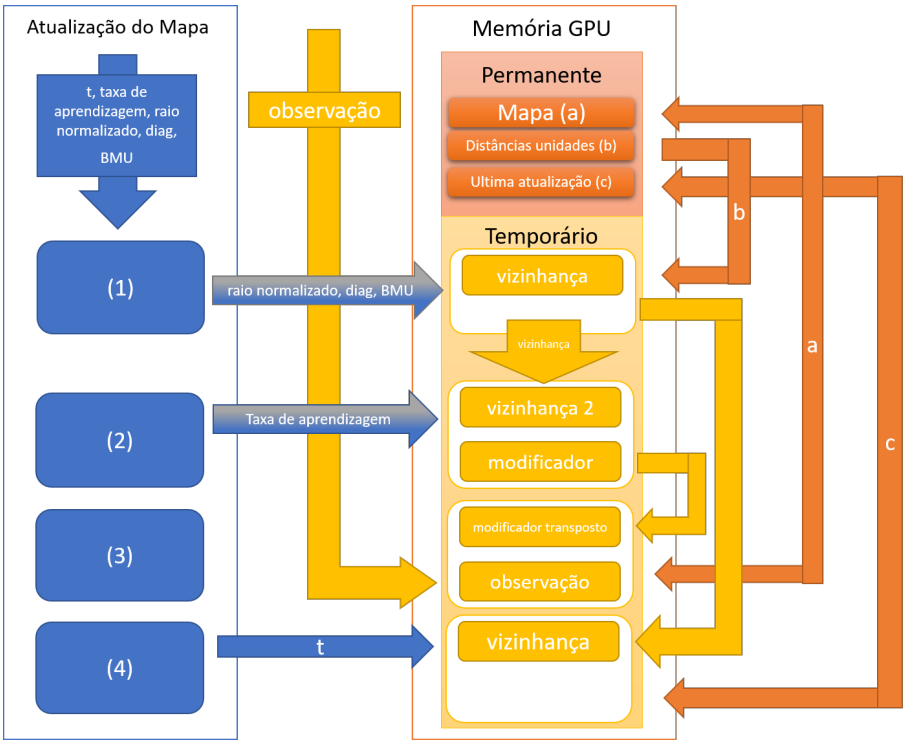


Figura 3.10: Transferências de memória durante a execução da atualização do mapa. O estágio (1) representa o cálculo da função de vizinhança. O estágio (2) respresenta o cálculo da magnitude de modificação para cada unidade. O estágio (3) representa a atualização do mapa. As letras representam as interações com as informações alocadas de forma permanente no GPU: (a) mapa, (b) distâncias entre unidades, (c) última atualização de cada unidade.

Listagem 3.5: *Kernel* Marrow para calculo do erro topológico

```
float compute_topological_error(
    int bmu,
    array<float, lines * columns> &distances
){
    int second_bmu = get_second_bmu(bmu, distances);
    topological_error_queue[t % window_size] =
        is_adjacent(bmu, second_bmu);
    scalar<float> sum_topological_error_queue =
        reduce<plus<float>>()(topological_error_queue);

    return sum_topological_error_queue.value() /
        (float) window_size;
}

int get_second_bmu(
    int bmu,
    array<float, lines * columns>& distances
) {
    int bmu_distance = distances[bmu];
    distances[bmu] = INT_MAX;
    int second_bmu = get_bmu(distances);
    distances[bmu] = bmu_distance;
    return second_bmu;
}


int is_adjacent(int bmu, int second_bmu)
{
    if(unit_distances[bmu][second_bmu] > 1)
        return 0;
    else return 1;
}
```

3.2.4.1 Erro Topológico

A listagem 3.5 descreve o processamento do erro topológico. Este erro topológico é calculado por verificar se a 1ª e a 2ª BMU são adjacentes para uma determinada observação. Esse resultado é utilizado numa média com os resultados anteriores (normalmente utilizando uma janela) para obter o erro topológico. A equação 3.1 formaliza esse cálculo. Assim, $\overline{te}$ representa o erro topológico, bmu_1 representa a 1ª BMU, bmu_2 representa a 2ª BMU e $window\_size$ o tamanho da janela mencionado na tabela 3.1.

$$\overline{te} = \sum 1_{\{\|bmu_1 - bmu_2\| \leq 1\}} / window\_size \tag{3.1}$$

O método que cálcula o erro topológico tem como argumentos a BMU atual e um *array* com as distâncias da BMU a cada unidade do mapa. Esses dois parâmetros são assim usados para calcular qual é a segunda BMU. De seguida verifica-se se a 1ª e a 2ª BMU são adjacentes, colocando esse resultado no vector circular *topological_error_queue*.

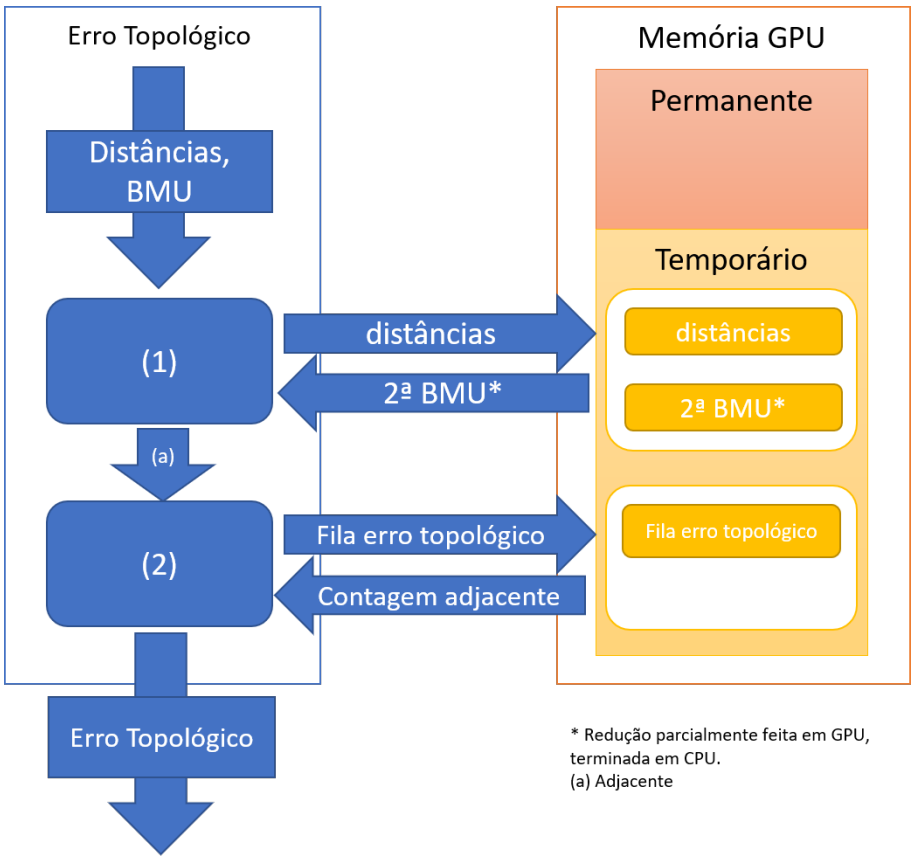


Figura 3.11: Transferências de memória ao processar o Erro Topológico.

Aplica-se uma redução de soma a essa fila, sendo o seu resultado dividido pela dimensão da janela, obtendo assim o erro topológico médio do mapa.

Este método faz uso de dois *kernels* OpenCL. O método para obter a segunda BMU utiliza o método *get_bmu* já descrito anteriormente na secção 3.2.2. É ainda executada uma redução de *array* para um escalar na linha 8 (e 9), disponível na listagem A.7 no apêndice A.

A figura 3.11 ilustra as transferências de memória no cálculo do erro topológico. Como vemos na ilustração, para o primeiro *kernel* OpenCL, é necessário enviar o *vector* de distâncias para o GPU, e retornar a redução parcial para o CPU para ser terminada. Para o segundo *kernel* é necessário enviar a fila de erro topológico, e após a operação de redução, a contagem de adjacências.

É de notar, porém, que esta implementação do erro topológico não é a melhor solução possível devido à forma como a segunda BMU é obtida. A implementação atual envolve alterar um valor único na matriz de distâncias, forçando a uma nova transferência do contéudo do vetor de distâncias para a memória do GPU. Uma possível alternativa seria poder especificar na *framework* Marrow se queremos o n menor (ou maior) elemento da colecção passada como argumento á função de redução. Outra alternativa é o uso de uma função de ordenação implementada em GPU que retorna os elementos (ou os seus

índices) por ordem crescente ou decrescente.

3.2.4.2 Erro Médio de Quantização

Na listagem 3.6 está a implementação do kernel Marrow para o cálculo do Erro Médio de Quantização (Equação 2.8). Como foi visto na listagem mencionada, começa-se por processar o erro de quantização atual a partir da observação atual, mas também a partir da BMU atual e do valor de *omega* ($|\Omega|$, na equação 2.8), ou seja, o valor máximo esperado para a distância euclidiana entre dois pontos. Tendo em conta que trabalhamos num espaço normalizado entre 0 e 1, podemos assumir que esse valor pode ser o mesmo que o número de características ($feature\_size$). Esse cálculo é efetuado por obter o quadrado da distância entre a BMU e a observação atual. O resultado é depois dividido por *omega* para obter o erro de quantização. O erro de quantização é depois colocado num *array* circular com $window\_size$ de dimensão, a partir do qual se calcula o erro de quantização médio.

Neste processo são gerados 3 *kernels*. O primeiro, é definido pelo código Marrow entre as linhas 6 e 8 onde se efetua a sua composição (resultado visível na Listagem A.10, no apêndice A). O resultado desse *kernel* é depois atribuído ao segundo *kernel* (Listagem A.7, Apêndice A) que faz a soma de todos os resultados. A esse resultado é aplicada uma operação de raiz quadrada e aplicada a divisão por *omega*. O terceiro *kernel* é uma repetição do segundo *kernel*, mas com a fila de erros de quantização de iterações anteriores (com dimensão do tamanho da janela temporal).

O cálculo do erro médio de quantização recebe do CPU informações como a última observação, a ultima BMU e o valor de *omega*. Este processo aproveita o facto do Mapa estar já disponível no GPU. A figura 3.12 detalha mais esse processamento. Na primeira etapa, os dois primeiros *kernels* OpenCL já mencionados são executados. Para esse fim é necessário transmitir informações como a última observação processada e a BMU para o GPU (assim como os *kernels*). No fim da operação, é trazido o resultado da redução de volta para o CPU. Na segunda etapa do processamento, é transmitida para o GPU a fila com os erros de quantização dimensão $window\_size$. É depois retornado do GPU o somatório dessa fila.

3.2.4.3 Utilidade Média de Neurónio

Na listagem 3.7 está a implementação do método Marrow para o cálculo do utilidade média de neurónio (Equação 2.10). Ao calcular utilidade de neurónio, obtêm-se a diferença de tempo desde a última atualização para cada neurónio e verifica-se se a última vez que esse neurónio foi atualizado foi há mais iterações do que a dimensão da janela de histórico ($window\_size$). Depois conta-se quantos neurónios foram atualizados recentemente e divide-se esse valor pelo número de unidades no mapa. De seguida coloca-se esse resultado num *array* circular com o tamanho da janela. Esse *array* é depois utilizado para calcular a média das ultimas $window\_size$ iterações, obtendo assim a utilidade média de neurónio.

Listagem 3.6: *Kernel* Marrow que calcula o erro médio de quantização.

```
float process_quantization_error (
    array<float, feature_size>& observation,
    int bmu,
    float omega)
{
    auto som_bmu = som_map[bmu];
    auto sub = observation - som_bmu;
    auto eq_map = marrow::pow(sub, 2);
    scalar<float> eq_sum = marrow::reduce<plus<float>>()(eq_map);

    return marrow::sqrt(eq_sum.value()) / omega;
}

//quantization error
float omega = feature_size * 1.0f;
this->last_omega = omega;
float quantization_error =
    process_quantization_error(*obs, bmu, omega);

//compute average quantization error
quantization_error_queue[t % window_size] =
    quantization_error;
scalar<float> average_quantization_error2 =
    reduce<plus<float>>()(quantization_error_queue);
float average_quantization_error =
    average_quantization_error2.value() / window_size;
```

Este método gera 3 *kernels* OpenCL. O primeiro *kernel* (listado no apêndice A, listagem A.8), calcula quantas iterações passaram desde a última atualização de cada neurónio. O segundo *kernel*, na listagem A.9 do apêndice A, efetua a soma de todos esses valores, cujo resultado é depois dividido pelo número de unidades do mapa para obter a utilidade de neurónio. O terceiro *kernel*, na listagem A.7 do apêndice A, faz a soma dos valores de utilidade de neurónios anteriores (dentro de uma determinada janela temporal), cujo resultado é depois dividido pelo tamanho da janela para obter a utilidade média de neurónios.

O cálculo da utilidade média de neurónio aproveita o fato de que os dados relacionados com a última atualização de cada uma das unidades estarem já armazenados na memória do GPU. O GPU recebe ainda o parâmetro t do a partir do CPU. A figura 3.13 detalha mais as transferências nesse processo. Na primeira etapa, consistindo no processamento a utilidade de neurónio na última iteração, é passado para o GPU um *kernel* composto nesta fase juntamente com o parâmetro t. Essa operação é terminada com um somatório, cujo resultado é retornado ao CPU. A primeira etapa termina com o cálculo da utilidade de neurónio, que é passado à segunda etapa. Esta segunda etapa adiciona a utilidade de neurónio mais recente à fila de utilidades de neurónio e procede em enviar essa fila para o GPU, onde é efetuada um segundo somatório, que é de seguida retornado

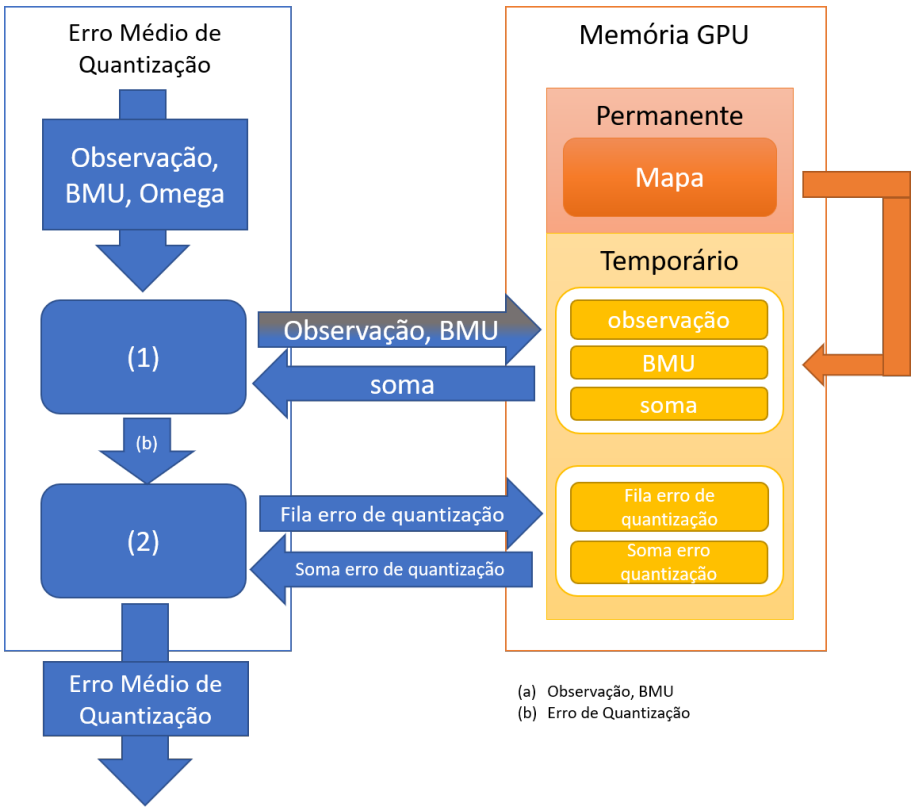


Figura 3.12: Transferências de memória ao processar o Erro Médio de Quantização.

Listagem 3.7: *Kernel* Marrow que calcula a utilidade média de neurónio.

```
float process_neuron_utility(int t,
    const array<int, lines * columns>& last_update)
{

    auto un_tmp = - last_update + t;
    auto un_tmp2 = conditional(un_tmp <= window_size, 1, 0);
    scalar<int> un_sumation = reduce<plus<int>>()(un_tmp2);
    float neuron_utility = ((float)un_sumation.value())
        / ((float)unit_count);
    return neuron_utility;

}

float neuron_utility = process_neuron_utility(t, t_k_update);
neuron_utility_queue[t % window_size] = neuron_utility;

//compute average neuron utility
scalar<float> neuron_utility_sum =
    reduce<plus<float>>()(neuron_utility_queue);
float average_neuron_utility = neuron_utility_sum.value()
    / window_size;
```

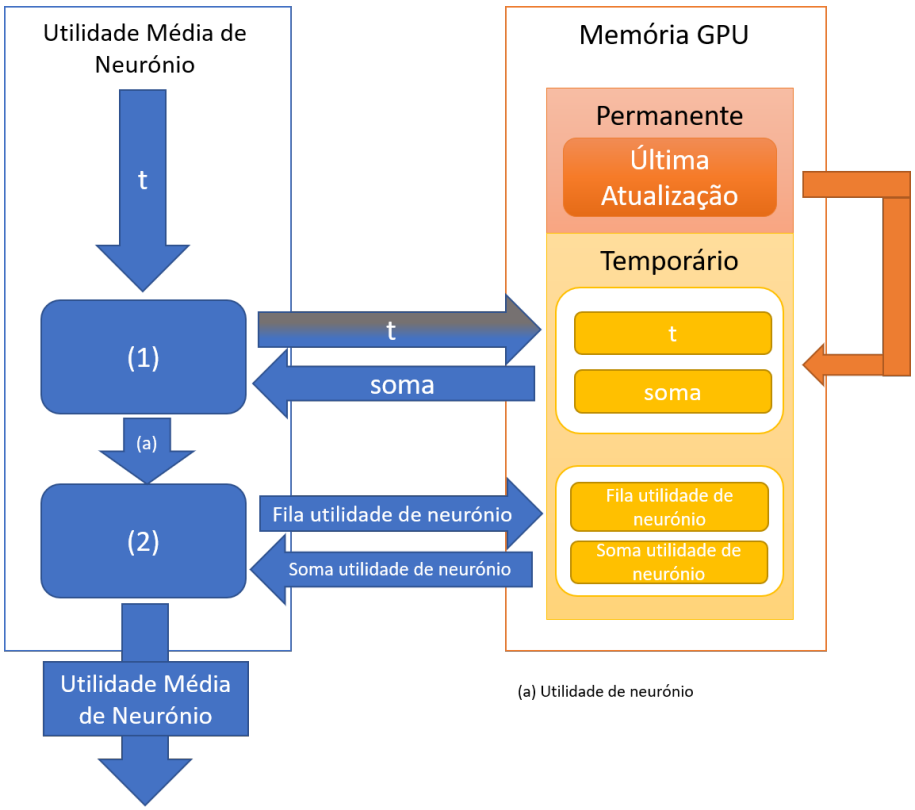


Figura 3.13: Transferências de memória ao processar a Utilidade Média de Neurónio.

ao CPU, para ser calculada a utilidade média de neurónio.

3.3 Optimizações ao Marrow

Sendo o Marrow uma plataforma ainda em desenvolvimento, existiam ainda alguns aspetos que no início desta dissertação não correspondiam exatamente às necessidades presentes.

3.3.1 Redução de Matriz para Array

Um desses aspetos era a implementação de uma operação em particular: a operação de redução de uma matriz a um *array*. A implementação original da operação tratava cada linha da matriz como um array individual, invocando um novo *kernel* OpenCL para cada linha, tendo posteriormente o objetivo de recuperar apenas um único valor de cada *kernel*. Devido ao custo de comunicação entre o CPU e o GPU, tal operação é prejudicial para performance do UbiSOM em GPU.

Com o objetivo de se resolver esse problema, implementou-se o *kernel* conceptualizado no fluxograma da figura 3.14. O *kernel* efetua somas aos pares, dentro de cada linha da matriz, reduzindo a cada iteração o número de valores a somar a metade. Quando finalmente sobra apenas um único valor, essa operação para e retorna o resultado para

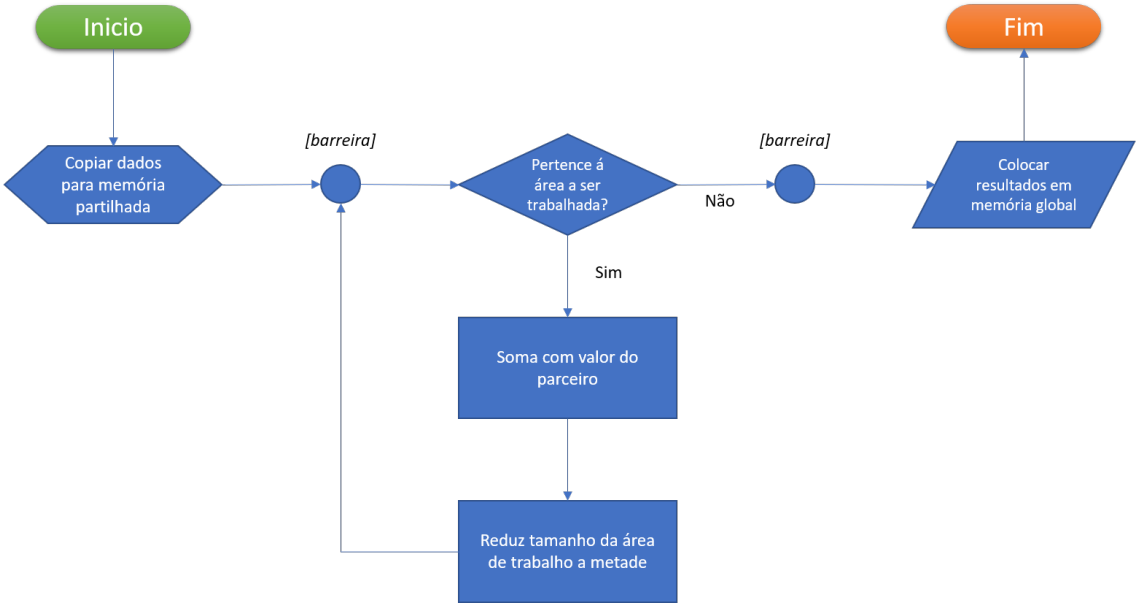


Figura 3.14: Conceptualização do *Kernel* Implementado

a posição do *array* desejada. Este progresso está ilustrado na figura 3.15. Nessa figura, onde cada quadrado numerado representa uma *thread*, sendo o número o seu identificar, é possível notar que em cada iteração (it), corta-se o conjunto de *threads* a metade antes de cada soma. Eventualmente, quando apenas sobra uma *thread*, essa *thread* é excluída e o processo termina.

Para além disso, o *kernel* é capaz de dividir cada linha da matriz por *work-groups* diferentes nos casos em que a linha não cabe dentro do mesmo *work-group*, efetuando uma soma atómica na posição de memória global com o resultado no fim da soma de cada segmento da linha.

O *kernel* descrito encontra-se nas listagens III.1, III.2 e III.3 no Anexo III.

Para testar esse *kernel*, e comparar com a implementação existente no Marrow, foram concebidos alguns testes unitários. As listagens I.13 e I.14, visíveis no anexo I.2, mostram testes concebidos, para verificar a correcção do *kernel* em si e para diferenciar a performance média e o número de transferências das duas versões ao longo de 50 iterações. As listagens I.15 e I.16 contém os testes que testam o *kernel* para reduzir matrizes com um número de colunas que não sejam potências de 2 ou impares, respetivamente. Isto permite-nos testar a correcção do *kernel* para divididas pelos limites dos *work-groups*. A listagem I.17 contém um teste que testa a redução da matriz para números de colunas muito elevados (neste caso 1000), que têm que ser divididas por vários *work-groups* para efetuar a redução.

A tabela 3.3 mostra os resultados obtidos por esses testes. Como é observável, obteve-se um *speedup* superior a 1212 vezes em relação á implementação original. Para além disso, muitas das transferências anteriormente efetuadas são evitadas com a versão nova.

Este novo método tem assim uma consequência positiva que beneficia a performance

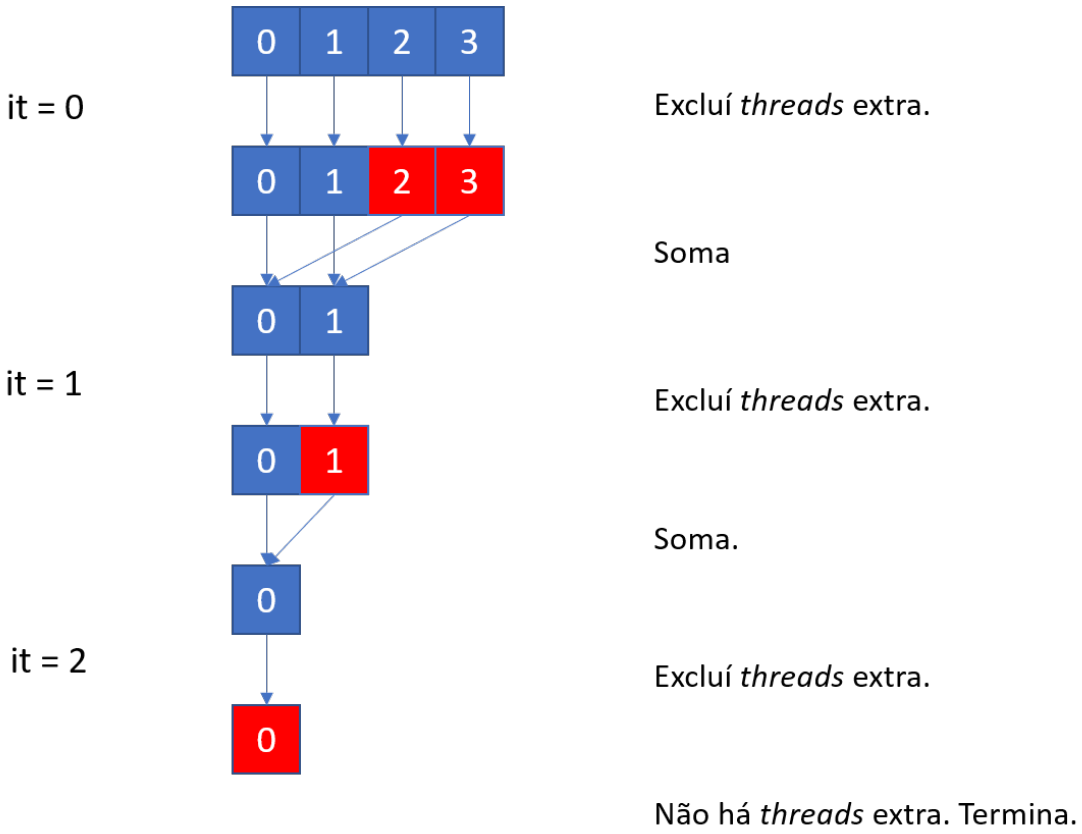


Figura 3.15: Conceptualização do processo de soma paralela

Tabela 3.3: Comparação entre implementação nova (OpenCL) e implementação original (Marrow).

	Tempo (ns)	Execuções de Kernel	Cópia para GPU	Cópia para CPU
OpenCL	161157	50	1	1
Marrow	195339534	40000	1	40000
Speedup	1212,107			

do algoritmo onde está inserido. A implementação original forçava a recuperação dos dados para o CPU, elemento a elemento, o que por si só, já prejudicava o desempenho do programa. Por outro lado, caso se desejasse operar sobre esses resultados, seria necessário transferir esses valores para o GPU, desperdiçando tempo nessa transferência. A nova implementação, por outro lado, só recupera o resultado da redução quando é absolutamente necessária transferi-la para o CPU, tendo os resultados imediatamente disponíveis para o *kernel* seguinte.

3.4 Comunicação

Para esta aplicação, foi escolhida uma arquitetura cliente-servidor. Assim, a aplicação UbiSOM em Marrow comunica através de um *socket* TCP, permitindo assim comunicação

de dados a partir de sistemas externos. No entanto, é necessário serializar e deserializar os dados de uma forma eficiente para comunicação em rede. Para esse fim, foi escolhida a tecnologia *Protocol Buffers* (Protobuf) da Google [62].

3.4.1 Mensagens

Nesta subsecção irá ser elaborado as mensagens a serem utilizadas para comunicar com o UbiSOM em Marrow. Cada mensagem terá um excerto de código de definição do Protobuf com uma explicação e exemplos em C++ para serializar ou deserializar a informação, de acordo com o tipo de mensagem.

Inicialização do Mapa A mensagem *Init*, cujos campos estão definidos na tabela 3.4 (Listagem IV.1 em Anexo), é enviada pelo cliente visualizador para inicializar o mapa. A mensagem tem vários campos opcionais, utilizadas para definir a parametrização do UbiSOM:

1. Taxa de aprendizagem inicial, sendo um campo do tipo *float* entre 0 e 1.
2. Taxa de aprendizagem final, sendo um campo do tipo *float* entre 0 e 1.
3. Raio de vizinhança normalizado inicial, sendo um campo do tipo *float* entre 0 e 1.
4. Raio de vizinhança normalizado final, sendo um campo do tipo *float* entre 0 e 1.
5. *Beta*, um campo do tipo *float* entre 0 e 1, utilizado no cálculo da função de deriva, para equilibrar os valores do erro médio de quantização e da utilidade média de neurónios.
6. Mapa inicial, uma lista de valores com o estado inicial do mapa.

Estes valores são todos opcionais. Caso não tenham sido preenchidos, valores por defeito serão atribuídos. No caso do mapa, se não existir nenhum elemento presente na lista, é inicializado um mapa novo de forma aleatória. Caso existam elementos nessa lista, a lista deve ter exatamente o $width * height * feature\_count$ elementos, sendo $width$ a largura do mapa, $height$ a altura do mapa e $feature\_count$ o número de dimensões de cada unidade.

O servidor devolve a mensagem *InitAck* posteriormente, caracterizada na tabela 3.5 (Listagem IV.2, em Anexo). Esta mensagem possuí um único campo booleano que indica se a operação foi bem-sucedida ou não.

Observações para treino A mensagem *Train*, caracterizada na tabela 3.6 (Listagem IV.3, em Anexo), permite comunicar ao UbiSOM em Marrow quais são as novas observações para serem treinadas pelo algoritmo. A mensagem possui a listagem *observations* com a lista de observações a comunicar. Esta lista deve ter uma dimensão múltipla ao número

Tabela 3.4: Campos da mensagem *Init*

Nome	Tipo	Lista	Obrigatório
initial_learning_rate	float	não	não
final_learning_rate	float	não	não
initial_normalized_radius	float	não	não
final_normalized_radius	float	não	não
beta	float	não	não
initial_map	float	sim	não

Tabela 3.5: Campos da mensagem *InitAck*

Nome	Tipo	Lista	Obrigatório
success	bool	não	sim

Tabela 3.6: Campos da mensagem *Train*

Nome	Tipo	Lista	Obrigatório
observations	float	sim	não

de características de cada observação. Esta mensagem é enviada pelo cliente que serve como fonte de dados.

Requisitar informações do Mapa Caso seja preciso requisitar o estado atual do Mapa, existe a mensagem *RequestMap*. Esta mensagem é uma mensagem simples e sem parametrização.

Como resposta à mensagem *RequestMap*, a mensagem *Map* é enviada pelo programa. Esta mensagem, caracterizada na tabela 3.7 (Listagem IV.4, em Anexo), é composta por um valor t, correspondente ao índice temporal do mapa na altura em que foi pedido o estado do mapa, pelos valores $average\_neuron\_utility$, contendo qual é a utilidade média dos neurónios para o índice t atual, $average\_quantization\_error$, contendo qual é o erro médio de quantização para o índice t atual, e $drift$, contendo qual o valor da função de deriva para o índice t atual. Contém também uma lista de *floats* com o estado do mapa para um determinado índice t.

Tabela 3.7: Campos da mensagem *Map*

Nome	Tipo	Lista	Obrigatório
t	int32	não	sim
average_neuron_utility	float	não	não
average_quantization_error	float	não	não
drift	float	não	não
map	float	sim	não

Tabela 3.8: Campos da mensagem *SetMap*

Nome	Tipo	Lista	Obrigatório
new_map	float	sim	não

Tabela 3.9: Campos da mensagem *SetMapAck*

Nome	Tipo	Lista	Obrigatório
success	bool	não	sim

Tabela 3.10: Campos da mensagem *LastBmu*

Nome	Tipo	Lista	Obrigatório
x	int32	não	sim
y	int32	não	sim

Definir Estado do Mapa A mensagem SetMap, visível na tabela 3.8 (listagem IV.5, em Anexo), é enviada pelo cliente visualizador e serve para modificar o estado do mapa do UbiSOM quando necessário. Esta mensagem contém uma listagem *new_map*, com os valores do mapa, devendo ser preenchido da mesma forma que o campo *initial_map* da mensagem *Init*.

A esta mensagem é devolvida a mensagem *SetMapAck*, descrita na tabela 3.9 (listagem IV.6, em Anexo), que confirma ou não o sucesso da operação.

Obter última BMU A mensagem *GetLastBmu*, sem nenhuma parametrização, instruí ao servidor para devolver a ultima BMU obtida durante o treino. A essa mensagem, o servidor responde a mensagem *LastBmu*, descrita na tabela 3.10, possuindo um par de coordenadas da posição da BMU no mapa. Essas mensagens estão listadas na listagem IV.7, em Anexo.

Processar BMU para determinada observação Nas tabelas 3.11 e 3.12 (Listagem IV.8 em Anexo), vemos as mensagens *RequestBmuProcess* e *ProcessedBmu*. A primeira mensagem consiste num pedido para processar qual seria a BMU atual para a observação contida nela. Essa mensagem contém uma lista de valores correspondendo à observação desejada. Essa lista tem que ter um comprimento igual ao número de características. A segunda mensagem, como resposta à primeira, contém um par de coordenadas correspondente à BMU para a observação enviada.

Tabela 3.11: Campos da mensagem *Train*

Nome	Tipo	Lista	Obrigatório
observation	float	sim	não

Tabela 3.12: Campos da mensagem *SetMapAck*

Nome	Tipo	Lista	Obrigatório
x	int32	não	sim
y	int32	não	sim

Tabela 3.13: Campos da mensagem *Error*

Nome	Tipo	Lista	Obrigatório
error_type	int32	não	não
error_message	string	não	não

Mensagem de Erro As mensagens de erro servem para retornar erros ao cliente. Muitas vezes são utilizadas quando uma operação tem os argumentos inválidos, ou quando existe algum problema com o UbiSOM ou com o servidor. Na tabela 3.13 (Listagem IV.9 em Anexo), está visível a definição da mensagem de erro. Esta mensagem possui um campo *error_type* com um determinado código de erro e o campo *error_message*, com uma mensagem em texto a descrever o erro.

Capítulo 4

Avaliação

Neste capítulo serão discutidas as métricas de avaliação utilizadas para validar a implementação do UbiSOM e validar os ganhos de desempenho da implementação em GPU e qual a metodologia para obter os dados necessários. Falaremos ainda da parametrização do UbiSOM utilizada, quais os conjuntos de dados utilizados e qual o equipamento utilizado. Posteriormente será discutido o uso de testes unitários para validar a implementação. Efectuar-se-á ainda a análise dos resultados dos *datasets* utilizados. De seguida, serão comparado os tempos de execução entre o CPU utilizando 1, 2, ou 3 *threads* e os GPUs a serem utilizados. Após essa comparação, vai-se analisar o impacto dos *mini-batches* na correcção do algoritmo. Finalmente, serão resumidas as conclusões deste capítulo.

4.1 Métricas de Avaliação

Para avaliar o algoritmo implementado, serão verificados os aspetos da correção algorítmica e os ganhos obtidos em termos de velocidade (ou desempenho) da execução com esta implementação em GPU.

Para validar os aspectos de correcção algorítmica, vai-se procurar validar funcionalmente os vários componentes individuais do UbiSOM. Também serão comparados os resultados da implementação original do UbiSOM, com base na tese de Bruno Silva [69], com os resultados obtidos com esta implementação do UbiSOM em GPU. Por último, seão analisados alguns detalhes da implementação, como, por exemplo, o efeito do *multi-threading* e das dimensões do *mini-batches* no algoritmo.

Em relação ao desempenho, vai-se procurar comparar o tempo que demorou a treinar um mapa com um determinado *dataset*, o efeito do *mini-batch* no desempenho e o escalonamento com 1 ou mais *threads* e com o uso de GPU. Por fim procurar saber qual o tempo de execução médio para cada uma das etapas do algoritmo.

4.2 Metodologia

A avaliação do sistema implementado foi efetuada nas seguintes vertentes:

- Testes Unitários
- Validação dos Resultados
- Medição do Desempenho

Os testes unitários baseiam-se em simples testes à funcionalidade dos componentes da implementação, a ser elaborado na secção 4.3.1.

Quanto à validação de resultados, vai-se comparar visualmente os resultados da implementação do UbiSOM em CPU original e comparar os resultados com a versão em GPU, para cada *dataset* analisado. Para analisar o efeito do *multi-threading* em CPU, vai-se executar o algoritmo do UbiSOM Marrow em CPU com 1, 2 ou 4 *threads* no processador mencionado na secção 4.2.3. Para analisar o efeito dos *mini-batches* vai-se procurar obter os resultados com os tamanhos escolhidos de forma arbitrária de 1, 100 e 500, e fazer uma comparação visual.

Em relação à medição de desempenho, serão comparados o tempo que demorou a treinar um mapa com um determinado *data-set*, variando o tamanho do *mini-batch* (nomeadamente com tamanhos de 25, 50, 100, 250, 500 e 1000, escolhidos de forma arbitrária), para as diferentes implementações, nomeadamente CPU com 1 ou mais *threads* e GPU. Para além disso vai ser comparado o tempo médio que cada etapa do treino demora a efectuar.

Os *data-sets* utilizados para as vertentes de validação de resultados são discutidos na secção 4.2.2.

4.2.1 Parametrização do UbiSOM

Como recomendado na tese de Bruno Silva [69], o UbiSOM foi parametrizado com a taxa de aprendizagem a variar entre 0.1 e 0.08, o raio normalizado a variar entre 0.6 e 0.2, uma janela temporal com tamanho de 2000 e um valor de $\beta = 0.8$. O mapa é inicializado com um mapa previamente gerado, de forma aleatória, dentro de um hipercubo normalizado entre 0 e 1, utilizando o *script* Python no Anexo II, Listagem II.3. A dimensão do mapa utilizada é de 20 linhas por 40 colunas.

4.2.2 Conjuntos de Dados

Nesta secção serão elabadorados os conjuntos de dados (ou *datasets*) que serão utilizados para avaliar a implementação do UbiSOM em questão. Seguindo Bruno Silva[69] esses conjuntos de dados são:

- Irís

- Chain
- Complex
- Hepta
- Clouds

Todos esses *datasets* serão previamente normalizados com o uso do *MinMaxScaler* da biblioteca *sklearn* em Python (*script* em Anexo II, Listagem II.1).

4.2.2.1 Íris

O *dataset* Irís [74] é um dos *datasets* que vai ser testado no UbiSOM. Este *dataset* representa as dimensões das pétalas de várias espécies de flores do tipo Íris. Esse *dataset* possuí 4 caracteristicas:

- comprimento da sépala em centímetros
- largura da sépala em centímetros
- comprimento da pétala em centímetros
- largura da pétala em centímetros

Devido à sua dimensão reduzida (de 150 observações), será necessário expandir de forma uniforme o *dataset*. Para esse fim implementou-se um *script* em Python com o objectivo de efectuar essa expansão (Anexo II, Listagem II.2). A quantidade arbitrariamente selecionada é de 5000 observações.

Para além do Irís descrito acima, serão usadas ainda duas versões expandidas: uma versão com 10 características e outra com 16 características. Em relação à versão com 10 caracteristicas, foi gerada por repetir cada uma das características do Íris mais uma vez e por adicionar duas colunas aleatórias utilizando a função *rand*() de um *software* de folhas de cálculo como o LibreOffice Calc ou o Microsoft Excel. No caso da versão com 16 características, repetiu-se as características do Irís mais duas vezes, e adicionou-se 4 colunas aleatórias, com o mesmo processo.

O propósito principal deste grupo de *datasets* é validar os ganhos de performance obtidos por utilizar o GPU à medida que se vai aumentando o número de características.

4.2.2.2 Chain

O *dataset* Chain [69], foi outro *dataset* utilizado para validar o UbiSOM original.

Este *dataset* consiste num conjunto de 100.000 pontos tri-dimensionais que descrevem um par de anéis interligados (agregamentos), sendo utilizado originalmente para validar se os agregamentos eram detetáveis pelo uso da *u-matrix*. Nesta tese, irá ser utilizado para comparar a correção da implementação discutida em relação á implementação original.

4.2.2.3 Complex

O *dataset* Complex [69], foi outro *dataset* utilizado para validar o UbiSOM original.

Este *dataset* consiste num conjunto de 100.000 pontos bi-dimensionais que descrevem uma estrutura de *clusters* complexa, com sete *clusters* distintos, sendo utilizado originalmente para validar se os agregamentos eram detetáveis pelo uso da *u-matrix*. Nesta tese, irá ser utilizado para comparar a correção da implementação discutida em relação à implementação original.

4.2.2.4 Hepta

O *dataset* Hepta [69], foi outro *dataset* utilizado para validar o UbiSOM original.

Este *dataset* consiste num conjunto de 150.000 pontos tri-dimensionais que descrevem uma estrutura de *clusters* que muda abruptamente ao chegar ao instante $t = 100001$. Dos sete *clusters* Gaussianos iniciais, um deles desaparece. Este *dataset* tinha o propósito de avaliar como os diferentes algoritmos baseados no SOM reagem a mudanças repentinas na distribuição subjacente e para motivar a proposta da métrica de utilidade média de neurónio. Nesta tese, irá ser utilizado para comparar a correção da implementação discutida em relação á implementação original.

Ao contrário dos restantes *datasets*, este *dataset* vai utilizar uma janela de tamanho 1500 e um valor de β de 0.7.

4.2.2.5 Clouds

O *dataset* Clouds [69], foi outro *dataset* utilizado para validar o UbiSOM original.

Este *dataset* consiste num conjunto de 200.000 pontos bi-dimensionais que descrevem uma mudança gradual da estrutura de três *clusters* gaussianos (a mudança ocorre entre os instantes $t = 50001$ e $t = 150000$. Este *detaset* tinha o propósito de avaliar como os diferentes algoritmos baseados no SOM reagem a mudanças graduais na distribuição subjacente. Nesta tese, irá ser utilizado para comparar a correção da implementação discutida em relação á implementação original.

Ao contrário dos restantes *datasets*, este *dataset* vai utilizar uma janela de tamanho 1500 e um valor de β de 0.7.

4.2.3 Equipamento

Para efetuar os testes serão utilizadas duas máquinas distintas, descritas na tabela 4.1. Para os testes de CPU, será utilizado como referência a configuração nomeada "GTX 1060", com o processador AMD FX-4300.

Tabela 4.1: Configurações do equipamento a ser utilizado nos testes.

Máquina	CPU	Threads	RAM	GPU	HDD
GTX 1050	Intel Core i7-7700HQ @ 2.8 Ghz	8	8 GB DDR4	NVIDIA GTX 1050 4 GB	256 GB SSD
GTX 1060	AMD FX-4300 @ 3.8 Ghz	4	8 GB DDR3	NVIDIA GTX 1060 3 GB	1 TB HDD

4.3 Correcção dos Resultados

Nesta secção será abordada a correcção do algoritmo implementado em duas vertentes. A primeira vertente, dos testes unitários, vai demonstrar como as várias componentes do UbiSOM foram testadas para validar a sua correcção algorítmica. A segunda vertente foi testar o UbiSOM com os *datasets* mencionados na secção 4.2.2 e comparar os resultados com a implementação original.

4.3.1 Testes unitários

Para validar o algoritmo, foi necessária a conceptualização de diversos tipos de testes, seguido a metodologia de *Test Driven Development* (TDD). Essa metodologia começa por definir quais os requisitos do algoritmo em si, utilizando esses requisitos para a elaboração dos diversos testes, as quais o software deve corresponder. Esta metodologia de teste baseia-se em ciclos de desenvolvimento curtos, e entre cada ciclo, o *software* é validado para verificar se não existiu nenhuma regressão ao código já valido, e para validar o que foi adicionado.

Um componente importante de TDD é o uso de testes unitários. Estes testes unitários permitem-nos verificar se os diferentes componentes do UbiSOM funcionam correctamente. Com esse fim, procurou-se testar as seguintes funcionalidades da classe UbiSOM:

- Definir/Obter o estado do mapa
- Processamento da BMU
- Testar os *kernels*
- Testar métricas do UbiSOM
- Validar método de teste

Estes testes foram elaborados utilizando a *framework* GTest da Google. As listagens com os testes unitários mencionados estão disponíveis no Anexo I.

Definir/Obter o estado do mapa Para validar o processo de definição e de obtenção do estado do mapa, foi implementado o teste unitário visível na listagem I.1. Resume-se a um simples teste em que se insere um mapa com o método *set_map* e obtêm-se o mapa com o método *get_map*.

Processamento da BMU Para validar o processamento foram implementados os testes presentes nas listagens I.2 e I.3. Estes testes baseiam-se na execução do método *process_bmu* para mapas com 1 e 2 dimensões.

Testes ao 1º *Kernel* O primeiro *Kernel* do algoritmo consiste essencialmente na cálculo das distâncias entre cada unidade do mapa e a observação atual. Para efeitos de testes, expôs-se essa funcionalidade do método *map_distances*, e testou-se esse método para 1 e 2 dimensões. As listagens I.4 e I.5 ilustram esses testes.

Teste ao 2º Kernel O segundo *kernel* do algoritmo consiste essencialmente em obter qual foi a menor distância computada pelo primeiro *kernel*, utilizando o método *get_bmu*. A listagem I.6 ilustra o teste em questão.

Testes ao 3º Kernel O terceiro *kernel* é o responsável pela atualização do mapa, utilizando o método *update_map*. As listagens I.7 e I.7 demonstram esses testes, o primeiro para um mapa 3x3 e o segundo para um mapa 5x5.

Testes às métricas do UbiSOM Estes testes correspondem a validar o cálculo da utilidade de neurónios (Listagem I.9), da quantização de neurónios (Listagem I.10) e do erro topológico (Listagem I.11).

Teste ao método de treino O método de treino (*train*) é a interface principal do algoritmo para submeter novas observações. É a combinação dos testes mencionados anteriormente. Este método aceita uma lista de observações que vão provocar alterações no mapa. A listagem I.12 descreve o teste executado sobre esse método. Esse teste baseia-se em alimentar uma observação ao mapa, e validar o estado do mapa no fim desse treino.

4.3.2 Resultados dos *Datasets*

Nesta secção compara-se a correção algorítmica deste implementação do UbiSOM. Para isso, os mapas resultantes da implementação aqui discutida serão comparados com os resultados presentes nos trabalhos de Bruno Silva [67, 69]. Para distinguir entre o UbiSOM original e o UbiSOM implementado para esta dissertação, a implementação aqui descrita vai ser chamada de UbiSOM-Marrow.

4.3.2.1 Irís

A figura 4.1 demonstra a visualização da *u-matrix* e da matriz de contagens pelo algoritmo UbiSOM com um *batch-size* de 1. A figura 4.2 [67], mostra o mapa gerado pela execução do *dataset* Irís no algoritmo SOM de Kohonen. Considerando que o mapa utilizado no artigo original tinha dimensões diferentes (15x20) do mapa recomendado para o UbiSOM (20x40, o utilizado nesta dissertação) e que é utilizado o algoritmo original SOM, que

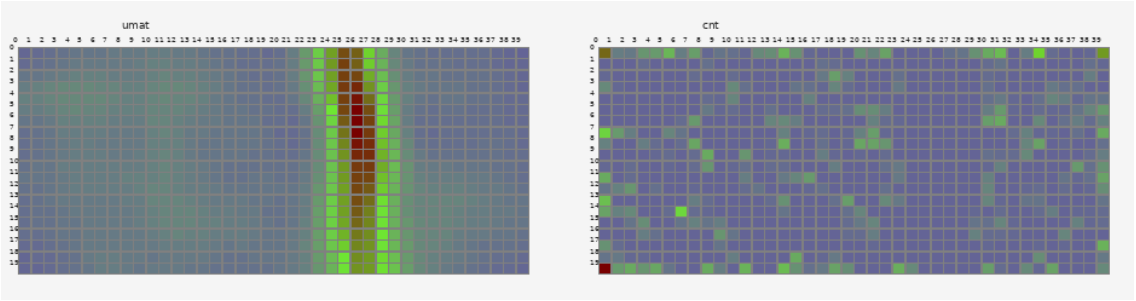


Figura 4.1: *U-Matrix* e Matriz de contagens do UbiSOM com *batch-size* = 1

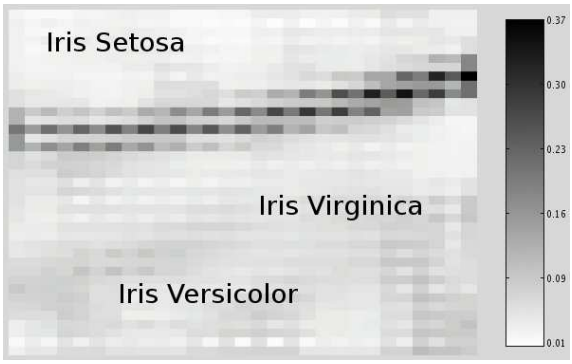


Figura 4.2: *U-Matrix* do *dataset* Irís com o algoritmo SOM [67].

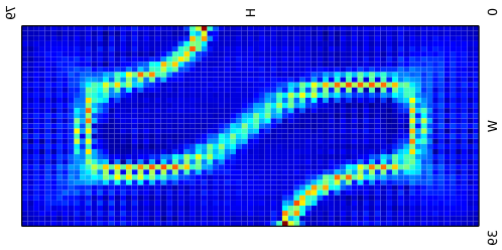


Figura 4.3: Resultado original do *dataset* Chain [69].

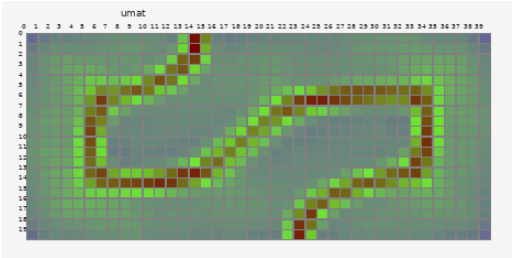


Figura 4.4: Resultado do *dataset* Chain no UbiSOM-Marrow com *batch-size* = 1

apesar de semelhante, é diferente do UbiSOM, é bastante visível as semelhanças entre os mapas gerados, mostrando que para fontes de dados uniformes e finitas, o UbiSOM e o SOM são equivalentes.

4.3.2.2 Chain

Nas figuras 4.3 e 4.4 vemos as semelhanças entre o resultado original do Chain visível na dissertação de Bruno Silva [69] e o processado pelo algoritmo UbiSOM-Marrow com *batch-size* de 1, mostrando que com essa configuração, e para este *dataset* é algoritmicamente equivalente.

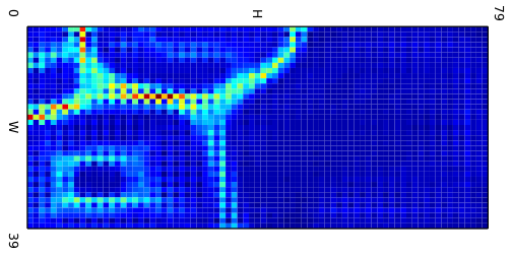


Figura 4.5: Resultado original do *dataset* Complex [69].

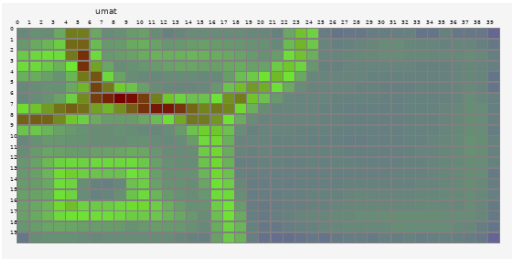


Figura 4.6: Resultado do *dataset* Complex no UbiSOM-Marrow com *batch-size* = 1

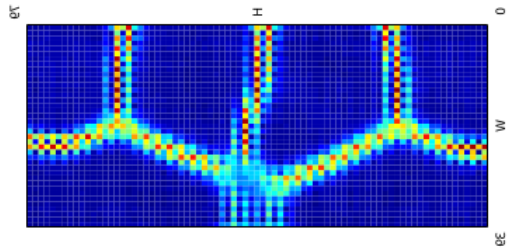


Figura 4.7: Resultado original do *dataset* Hepta [69].

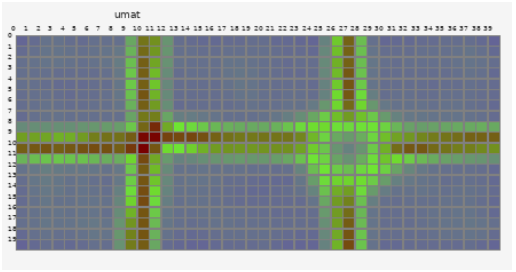


Figura 4.8: Resultado do *dataset* Hepta no UbiSOM-Marrow com *batch-size* = 1

4.3.2.3 Complex

Nas figuras 4.5 e 4.6 vemos as semelhanças entre o resultado original do Complex visível na dissertação de Bruno Silva [69] e o processado pelo algoritmo UbiSOM-Marrow com *batch-size* de 1, mostrando que com essa configuração, e para este *dataset* é algoritmicamente equivalente.

4.3.2.4 Hepta

Nas figuras 4.7 e 4.8 vemos as diferenças entre o resultado original do Hepta visível na dissertação de Bruno Silva [69] e o processado pelo algoritmo UbiSOM-Marrow com *batch-size* de 1. É possível notar que no caso desta fonte de dados não estacionária, a convergência foi feita de forma diferente do que o esperado, possívelmente devido a um estado do mapa inicial diferente do utilizado por Bruno Silva.

4.3.2.5 Clouds

Nas figuras 4.9 e 4.10 vemos as diferenças entre o resultado original do Clouds visível na dissertação de Bruno Silva [69] e o processado pelo algoritmo UbiSOM-Marrow com *batch-size* de 1. É de notar que o resultado do UbiSOM-Marrow é bastante diferente. Como anteriormente esta diferença pode ser provocada pela diferença no estado inicial do mapa em relação aos testes de Bruno Silva.

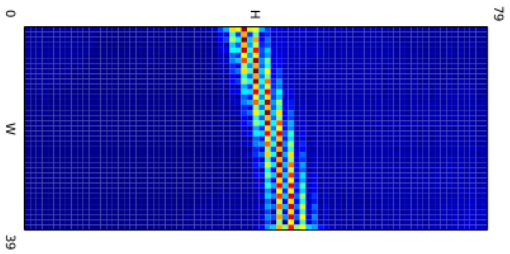


Figura 4.9: Resultado original do *dataset* Clouds [69].

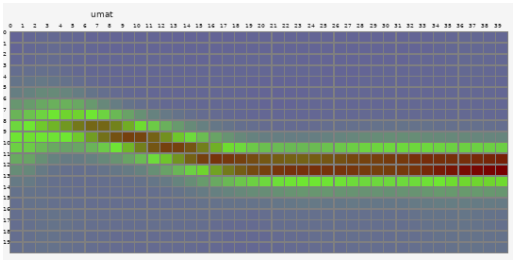


Figura 4.10: Resultado do *dataset* Clouds no UbiSOM-Marrow com *batch-size* = 1

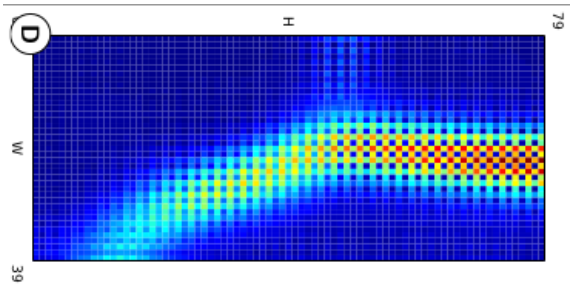


Figura 4.11: Mapa do UbiSOM original com o *dataset* Clouds no instante $t = 125000$.

4.3.2.6 Análise dos Resultados

Estes resultados aqui presentes mostram-nos que o algoritmo implementado é equivalente ao UbiSOM original com *datasets* cuja distribuição subjacente não sofre muitas mudanças. No entanto, as diferenças em relação aos *datasets* Hepta e Clouds sugeram que a adaptação para o GPU comporta-se de maneira diferente, possívelmente perdendo elasticidade, quando comparado com os resultados da implementação de Bruno Silva. Outro motivo para a diferença nos resultados pode ser o facto de não se usarem os estados iniciais usados por Bruno Silva. Mais trabalho sobre o que causa essas diferenças é necessário.

4.4 Desempenho

Estes testes de desempenho foram efetuados por executar o algoritmo UbiSOM com os conjuntos de dados já discutidos anteriormente na secção 4.2.2. Para cada um desses conjuntos de dados, foram variados vários elementos para além dos parâmetros intrinsecos ao conjunto de dados (como por exemplo, a quantidade de características), como o tipo de paralelização e o tamanho dos *batches*.

4.4.1 Comparação de desempenho versus CPU

As tabelas 4.2, 4.3 e 4.4 contém os tempos de execução para os *datasets* Íris, Íris 10 e Íris 16. Os gráficos nas figuras 4.12, 4.13 e 4.14 facilitam a visualização dos dados nessas tabelas.

Tabela 4.2: Tempos de execução do UbiSOM para o *Dataset* Irís em milisegundos.

batch-size	CPU 1T	CPU 2T	CPU 4T	GTX 1050	GTX 1060
25	117047	78470,7	51946,3	6572,32	11577,9
50	115744	75734,1	53753,7	6322,14	11422,6
100	115124	74096,5	49927,9	6113,15	11069,7
250	123207	74830,8	49107,6	6307,92	10673,5
500	115114	80210,8	39361	5955,88	10534,8
1000	115539	62736,7	39089,6	5854,59	10445,9

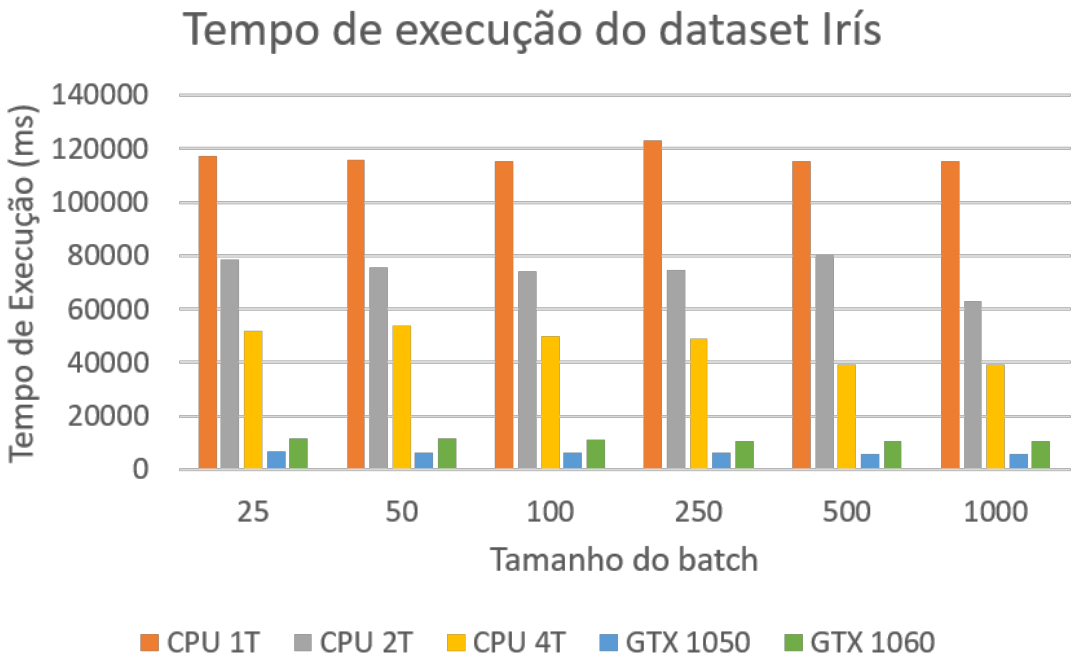


Figura 4.12: Gráfico da tabela 4.2

Tabela 4.3: Tempos de execução do UbiSOM para o Dataset Irís de 10 características em milisegundos.

batch-size	1T	2T	4T	GTX 1050	GTX 1060
1000	279368	163972	93803,1	5268,51	10150,9
500	279415	186548	106034	5520,59	11679,5
250	277036	189815	126414	5513,14	11077,1
100	276479	198839	111705	5548,27	12050,5
50	276379	188640	130540	5724,47	12512,8
25	277624	188931	130387	5966,63	12695,4

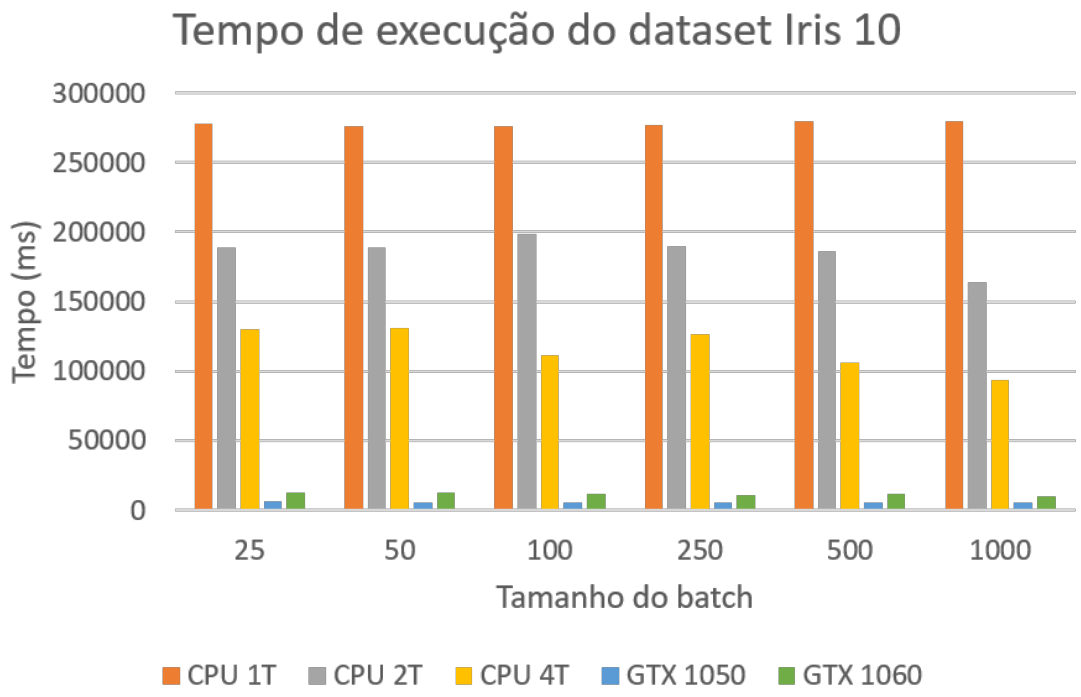


Figura 4.13: Gráfico da tabela 4.3

Tabela 4.4: Tempos de execução do UbiSOM para o Dataset Irís de 16 características em milisegundos.

batch-size	CPU 1T	CPU 2T	CPU 4T	GTX 1050	GTX 1060
25	428505	295109	185501	5924,9	12975,00
50	439437	295652	185307	5837,86	12599,40
100	430223	285348	179159	5597,81	12949,70
250	430643	278604	184600	5453,1	11068,00
500	427734	297015	146838	5327,2	11177,30
1000	428183	234775	162097	5339,59	10253,40

É possível observar que os GPUs têm uma vantagem muito significativa em relação aos CPUs na sua velocidade de processamento. Os gráficos 4.15, 4.16 e 4.17 mostram o *speedup* obtido entre as diferentes configurações de CPU (com 1, 2 e 4 Threads a correrem em paralelo) e os GPUs em questão (GTX 1050 e GTX 1060). Podemos ver que os resultados variam com um *speedup* de entre 10x (GTX 1060 na figura 4.15) e 77x (GTX 1050 na figura 4.17).

Há que notar um aspeto interessante nestes resultados. Um modelo de GPU superior (GTX 1060) consistentemente fica atrás de um modelo inferior (GTX 1050). Isso acontece porque o CPU que alimenta a GTX 1060 não consegue alimentar o GPU com trabalho suficiente para o manter completamente ocupado. A figura 4.18 ilustra esse facto. É possível ver que comparativamente, a GTX 1060 acaba por ter metade ou menos da taxa

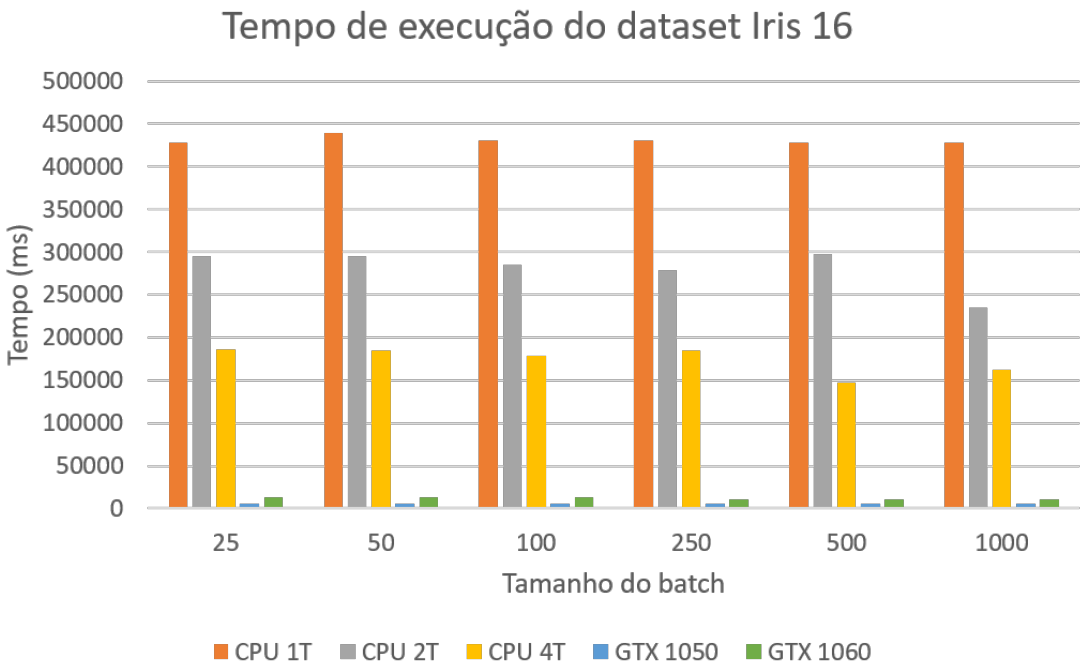


Figura 4.14: Gráfico da tabela 4.4

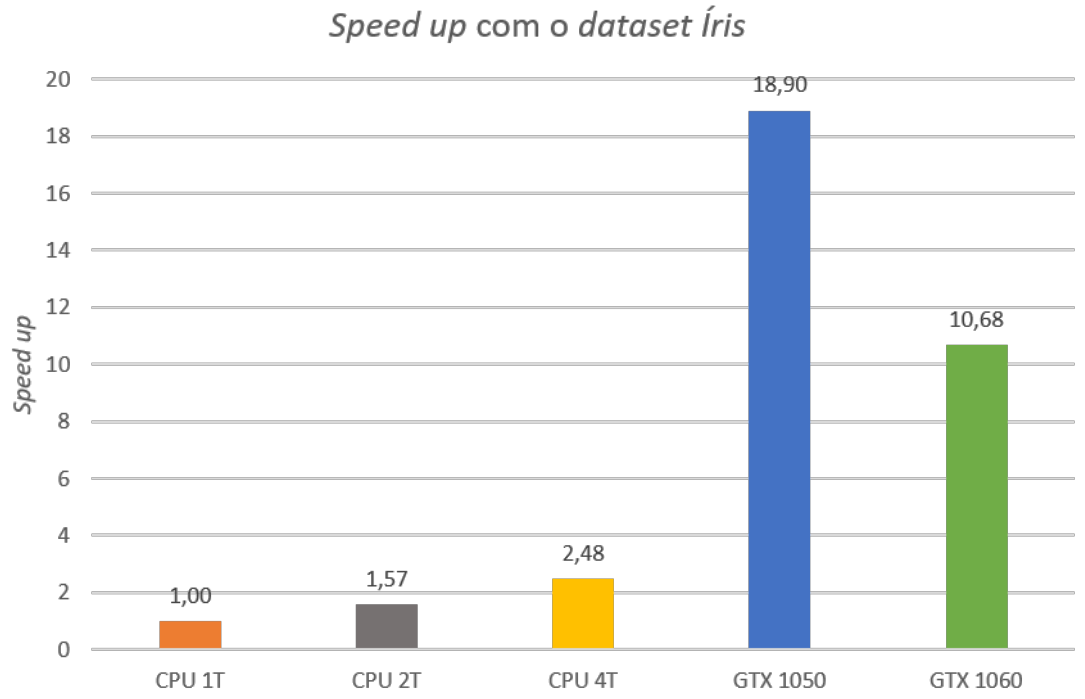


Figura 4.15: Gráfico com o *Speedup* médio com o *dataset* Irís

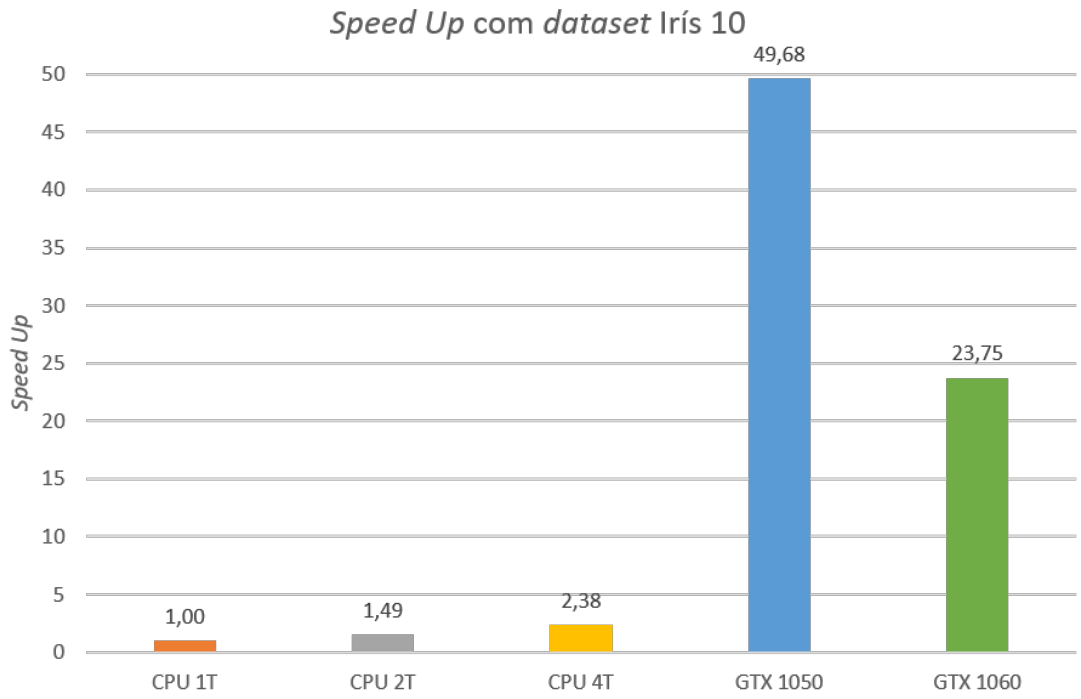


Figura 4.16: Gráfico com o *Speedup* médio com o *dataset* Irís de 10 características.

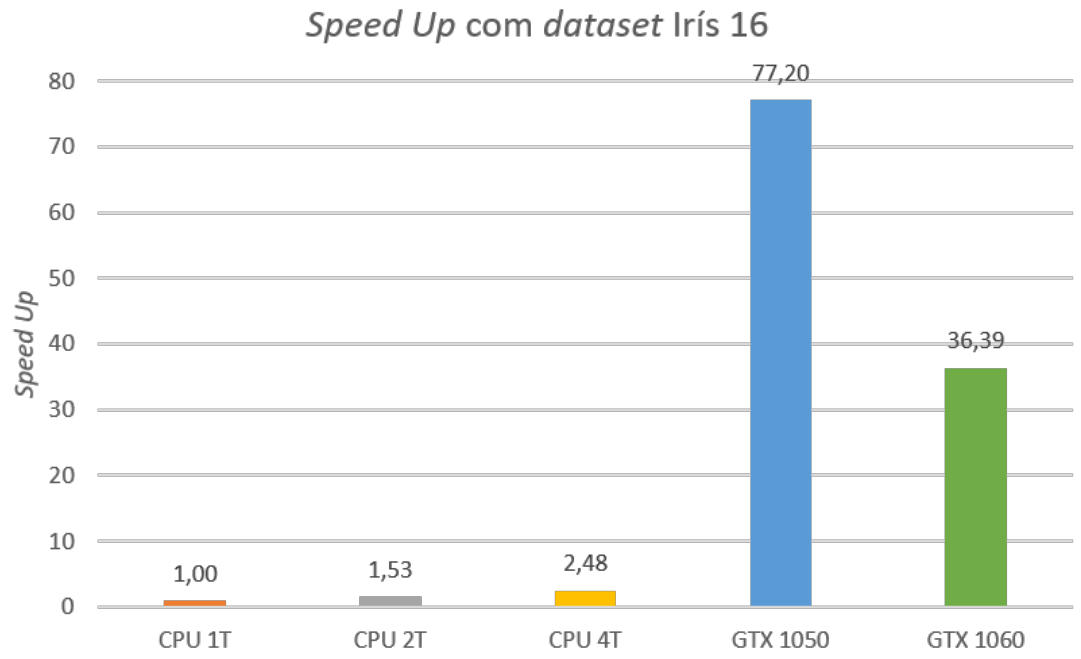


Figura 4.17: Gráfico com o *Speedup* médio com o *dataset* Irís de 16 características.

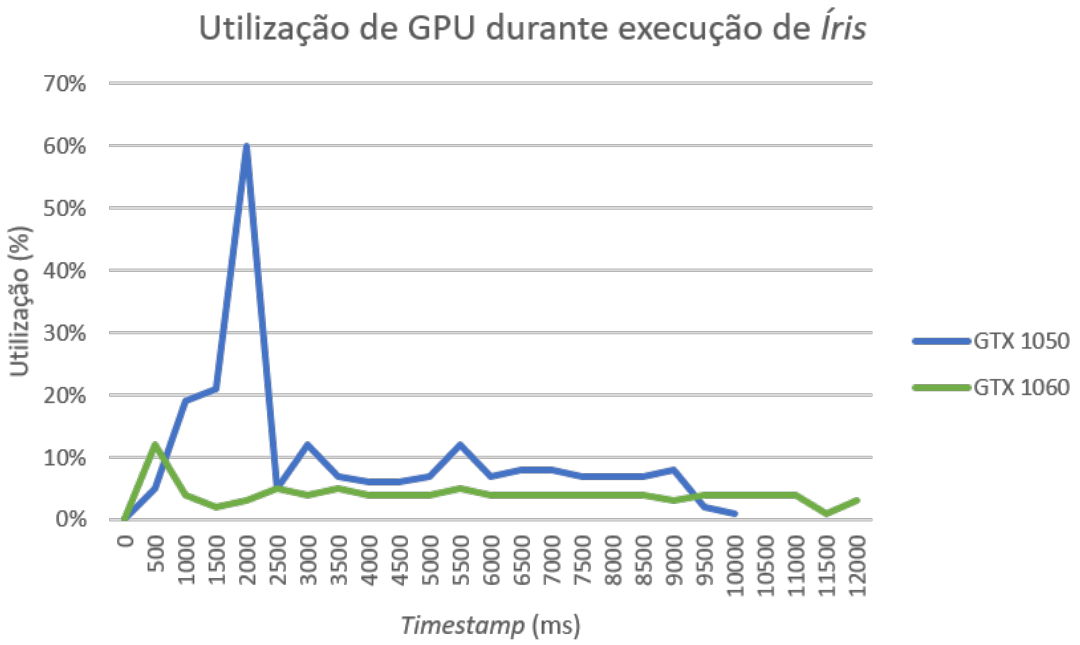


Figura 4.18: Utilização do GPU durante a execução do Íris Expandido.

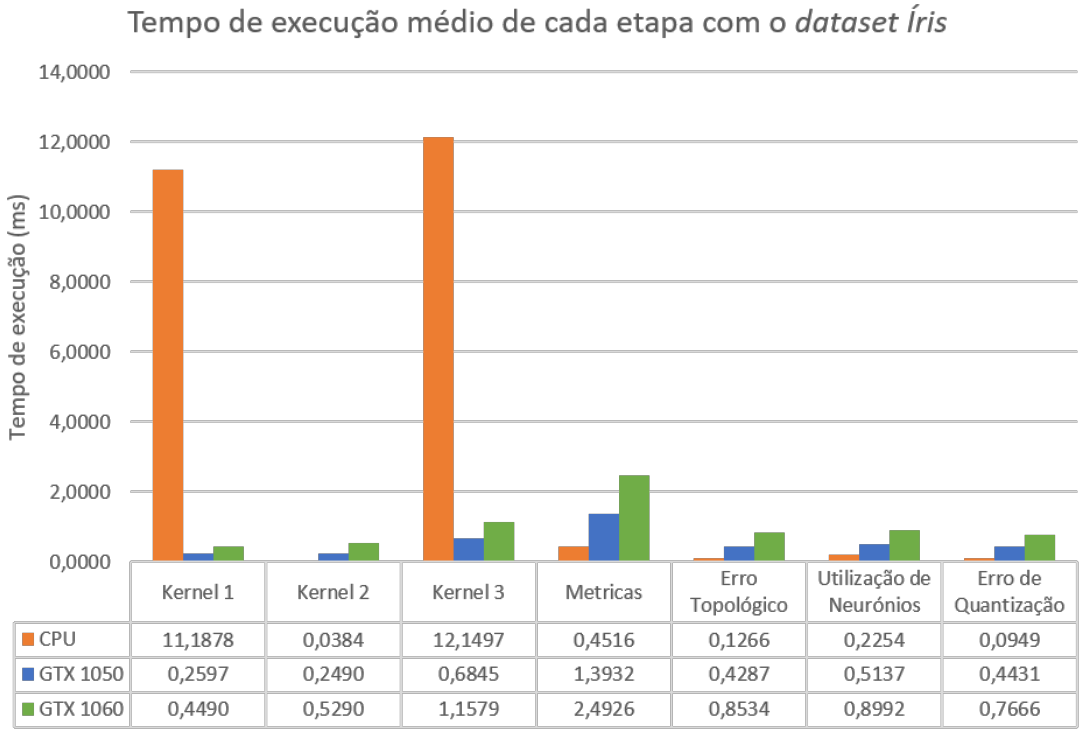
de utilização da GTX 1050, e acaba por terminar o trabalho mais tarde. Isto implica que mesmo assim, o CPU continua a ter uma influência bastante forte no processamento efetuado.

4.4.1.1 Análise Estágio a Estágio

Nas figuras 4.19, 4.21 e 4.23 mostram os tempos de execuções para cada um dos estágios do algoritmo. As figuras 4.20, 4.21 e 4.23 monstram os *speedups* correspondentes.

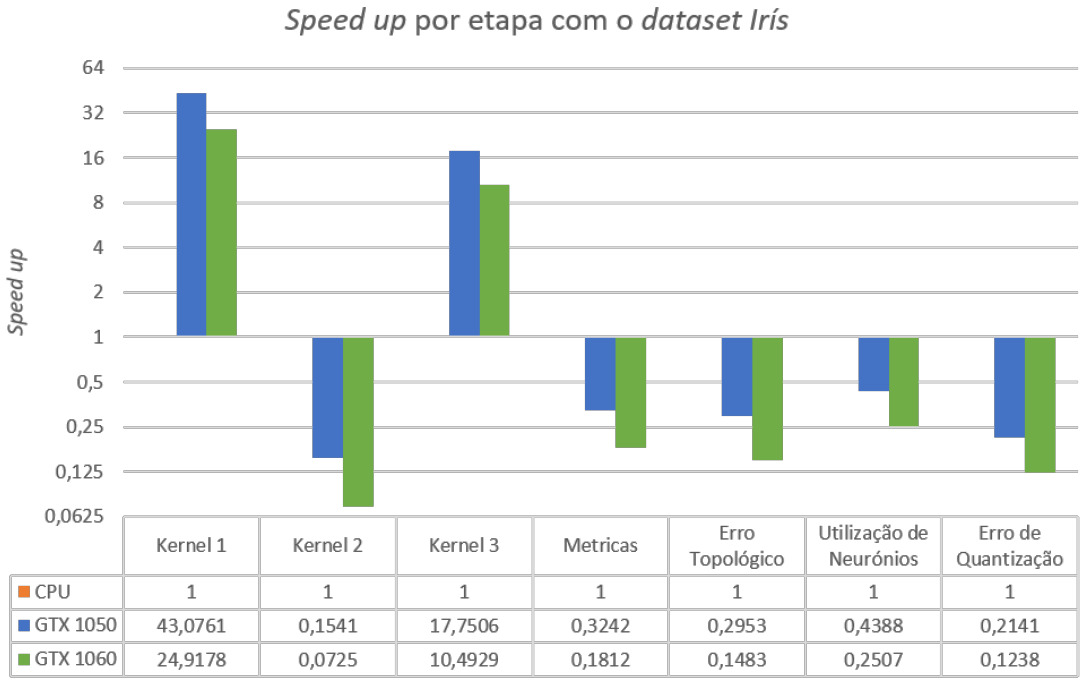
Como é possível observar, a maior parte do tempo que o CPU perde na execução do algoritmo é no estágios *Kernel 1* e *Kernel 3*, que é onde o GPU obtém a vantagem necessária para os *speedups* vistos anteriormente. Por exemplo, no caso mais extremo, o GPU obteve um *speedup* de 156 vezes no *Kernel 1* e de 84 vezes no *Kernel 2*.

No entanto é possível também notar que estágios como o *Kernel 2*, Erro Topológico, Utilização de Neurónios e Erro de Quantização foram mais rápidos a computar no CPU do que no GPU. O elemento que difere estes estágios dos estágios *Kernel 1* e *Kernel 3* é a existência de operações de redução de *argmin*, algo que efectivamente é calculado mais rápido em CPU que em GPU. Com isso em mente procurou-se testar a execução do algoritmo com a operação de redução executada no CPU. Os gráficos 4.25, 4.26 e 4.27 fazem a comparação entre a execução das reduções de *argmin* em GPU e CPU, utilizando a configuração GTX 1050. Como é visível, não existiu nenhuma vantagem ao executar essas operações em CPU. Isto ocorre devido à necessidade de recuperar as estruturas de dados presentes no GPU para o CPU, neutralizando a vantagem do CPU nessas operações.



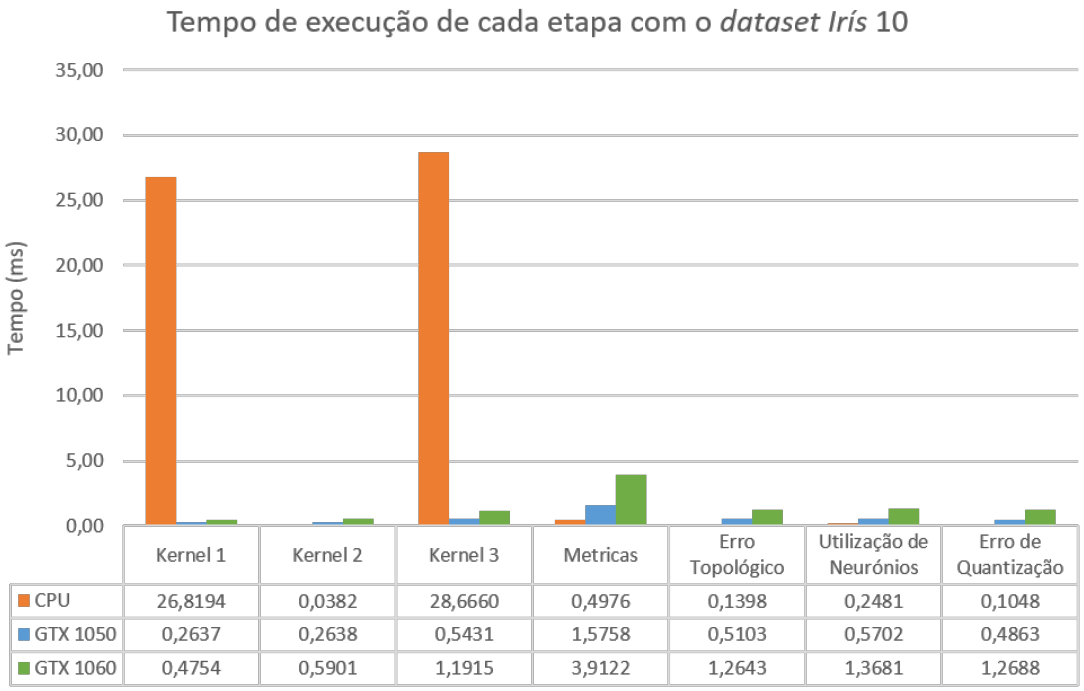
	Kernel 1	Kernel 2	Kernel 3	Metricas	Erro Topológico	Utilização de Neurónios	Erro de Quantização
CPU	11,1878	0,0384	12,1497	0,4516	0,1266	0,2254	0,0949
GTX 1050	0,2597	0,2490	0,6845	1,3932	0,4287	0,5137	0,4431
GTX 1060	0,4490	0,5290	1,1579	2,4926	0,8534	0,8992	0,7666

Figura 4.19: Gráfico com o tempo de execução médio por etapa com o *dataset* Irís



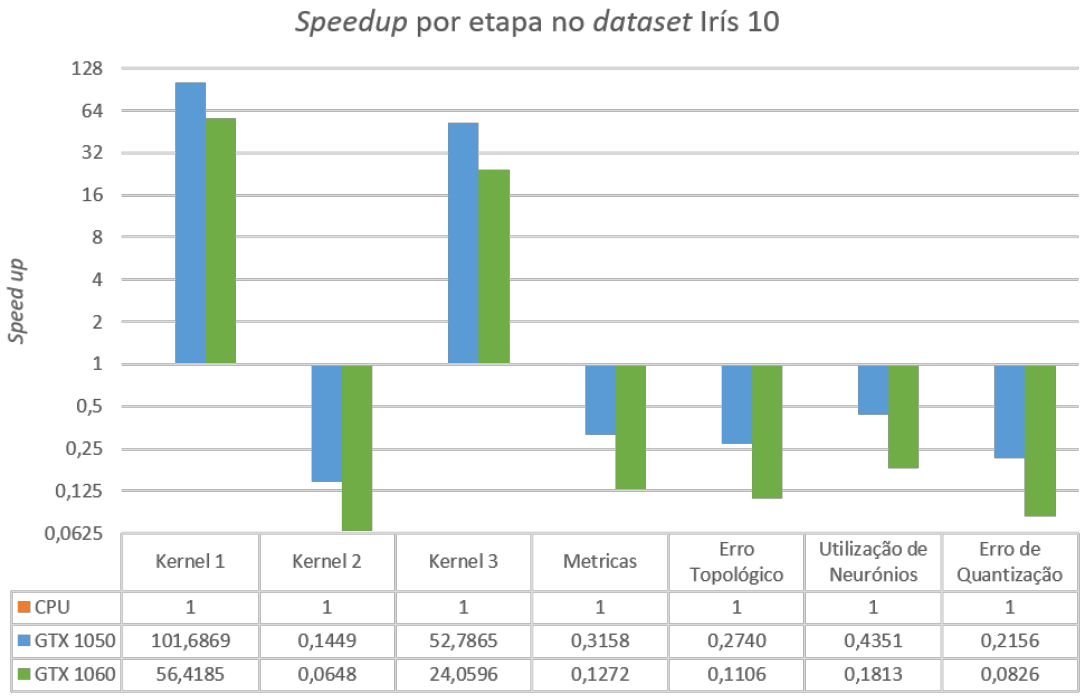
	Kernel 1	Kernel 2	Kernel 3	Metricas	Erro Topológico	Utilização de Neurónios	Erro de Quantização
CPU	1	1	1	1	1	1	1
GTX 1050	43,0761	0,1541	17,7506	0,3242	0,2953	0,4388	0,2141
GTX 1060	24,9178	0,0725	10,4929	0,1812	0,1483	0,2507	0,1238

Figura 4.20: Gráfico com o *Speedup* médio por etapa com o *dataset* Irís (Escala Logaritmica)



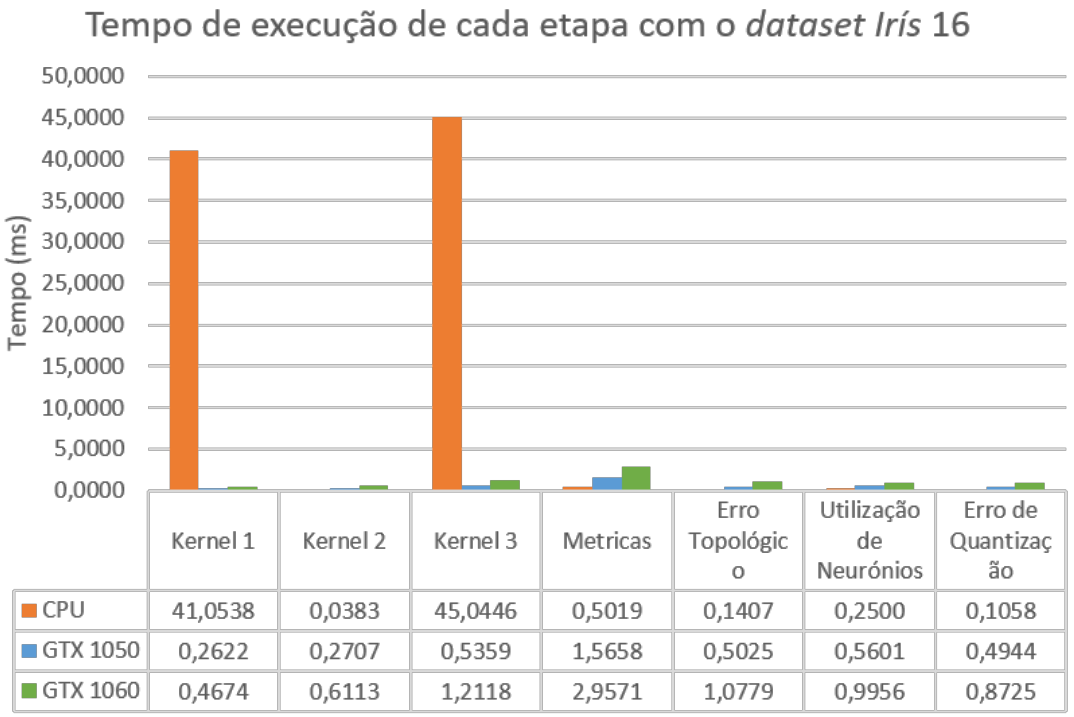
	Kernel 1	Kernel 2	Kernel 3	Metricas	Erro Topológico	Utilização de Neurónios	Erro de Quantização
CPU	26,8194	0,0382	28,6660	0,4976	0,1398	0,2481	0,1048
GTX 1050	0,2637	0,2638	0,5431	1,5758	0,5103	0,5702	0,4863
GTX 1060	0,4754	0,5901	1,1915	3,9122	1,2643	1,3681	1,2688

Figura 4.21: Gráfico com o tempo médio de execução por etapa com o *dataset* Irís de 10 características.



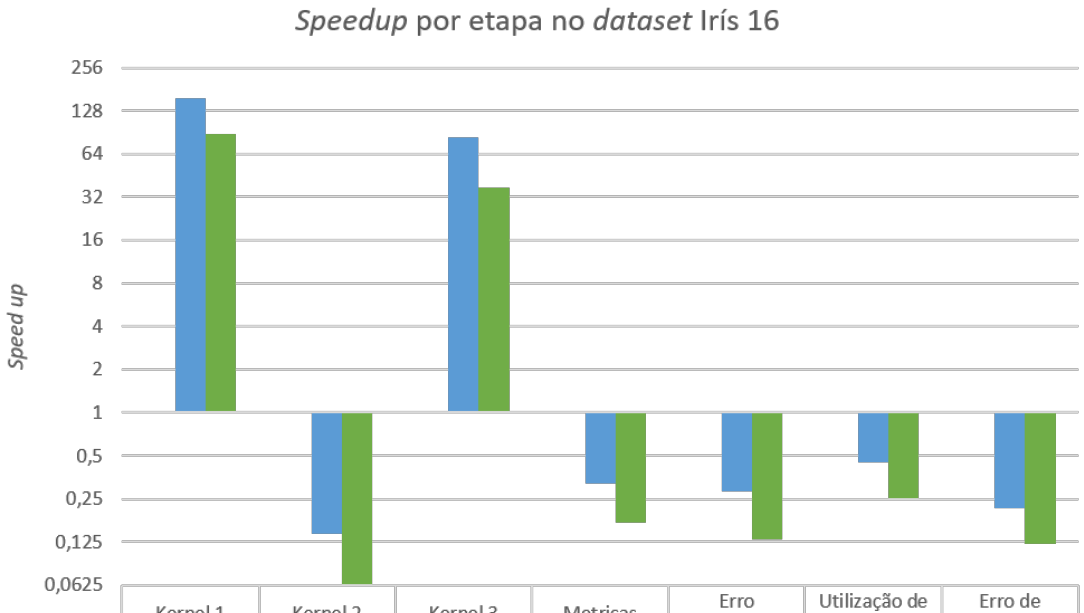
	Kernel 1	Kernel 2	Kernel 3	Metricas	Erro Topológico	Utilização de Neurónios	Erro de Quantização
CPU	1	1	1	1	1	1	1
GTX 1050	101,6869	0,1449	52,7865	0,3158	0,2740	0,4351	0,2156
GTX 1060	56,4185	0,0648	24,0596	0,1272	0,1106	0,1813	0,0826

Figura 4.22: Gráfico com o *speedup* médio por etapa com o *dataset* Irís de 10 características (Escala Logarítmica).



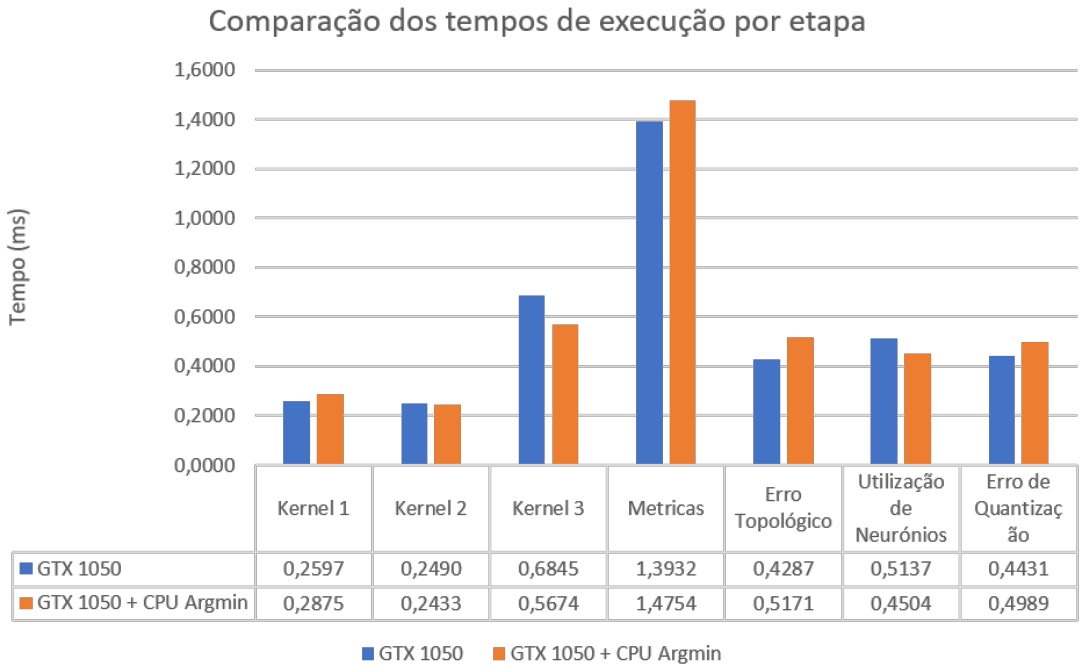
	Kernel 1	Kernel 2	Kernel 3	Metricas	Erro Topológico	Utilização de Neurónios	Erro de Quantização
CPU	41,0538	0,0383	45,0446	0,5019	0,1407	0,2500	0,1058
GTX 1050	0,2622	0,2707	0,5359	1,5658	0,5025	0,5601	0,4944
GTX 1060	0,4674	0,6113	1,2118	2,9571	1,0779	0,9956	0,8725

Figura 4.23: Gráfico com o tempo médio de execução por etapa com o *dataset* Irís de 16 características.



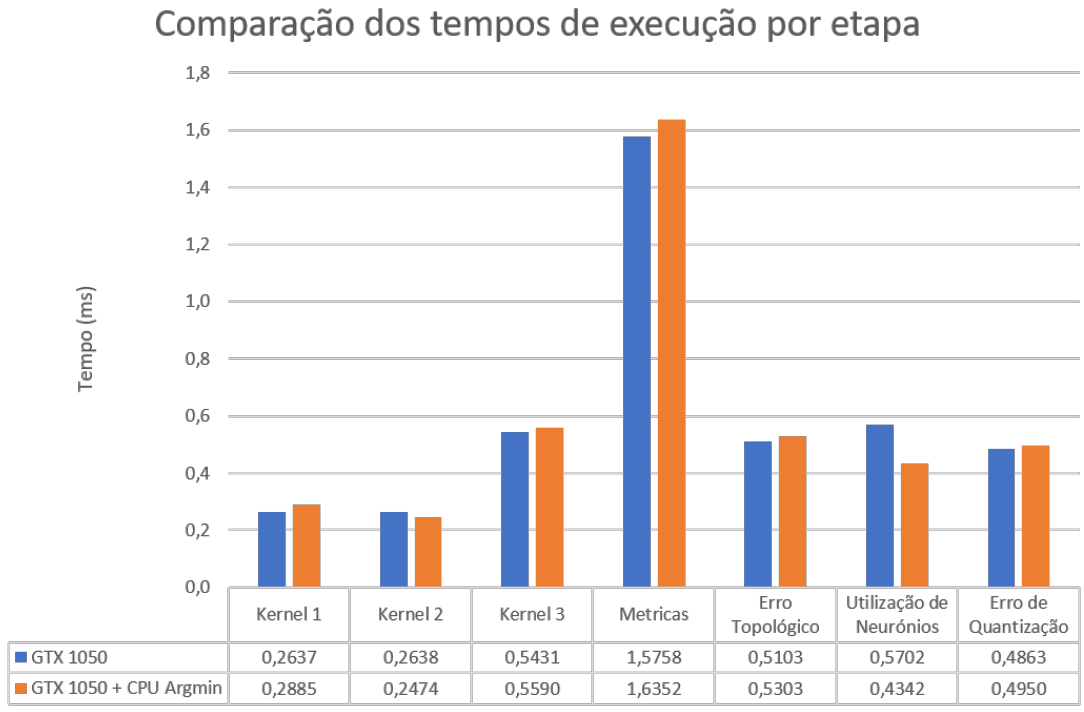
	Kernel 1	Kernel 2	Kernel 3	Metricas	Erro Topológico	Utilização de Neurónios	Erro de Quantização
CPU	1	1	1	1	1	1	1
GTX 1050	156,5623	0,1416	84,0484	0,3206	0,2801	0,4463	0,2140
GTX 1060	87,8259	0,0627	37,1723	0,1697	0,1306	0,2511	0,1213

Figura 4.24: Gráfico com o *speedup* médio por etapa com o *dataset* Irís de 16 características (Escala Logarítmica).



	Kernel 1	Kernel 2	Kernel 3	Metricas	Erro Topológico	Utilização de Neurónios	Erro de Quantização
GTX 1050	0,2597	0,2490	0,6845	1,3932	0,4287	0,5137	0,4431
GTX 1050 + CPU Argmin	0,2875	0,2433	0,5674	1,4754	0,5171	0,4504	0,4989

Figura 4.25: Gráfico com o tempo de execução médio de cada etapa com o *dataset* Irís, com as operação de *argmin* efetuadas no CPU.



	Kernel 1	Kernel 2	Kernel 3	Metricas	Erro Topológico	Utilização de Neurónios	Erro de Quantização
GTX 1050	0,2637	0,2638	0,5431	1,5758	0,5103	0,5702	0,4863
GTX 1050 + CPU Argmin	0,2885	0,2474	0,5590	1,6352	0,5303	0,4342	0,4950

Figura 4.26: Gráfico com o tempo de execução médio de cada etapa com o *dataset* Irís de 10 características, com as operação de *argmin* efetuadas no CPU.

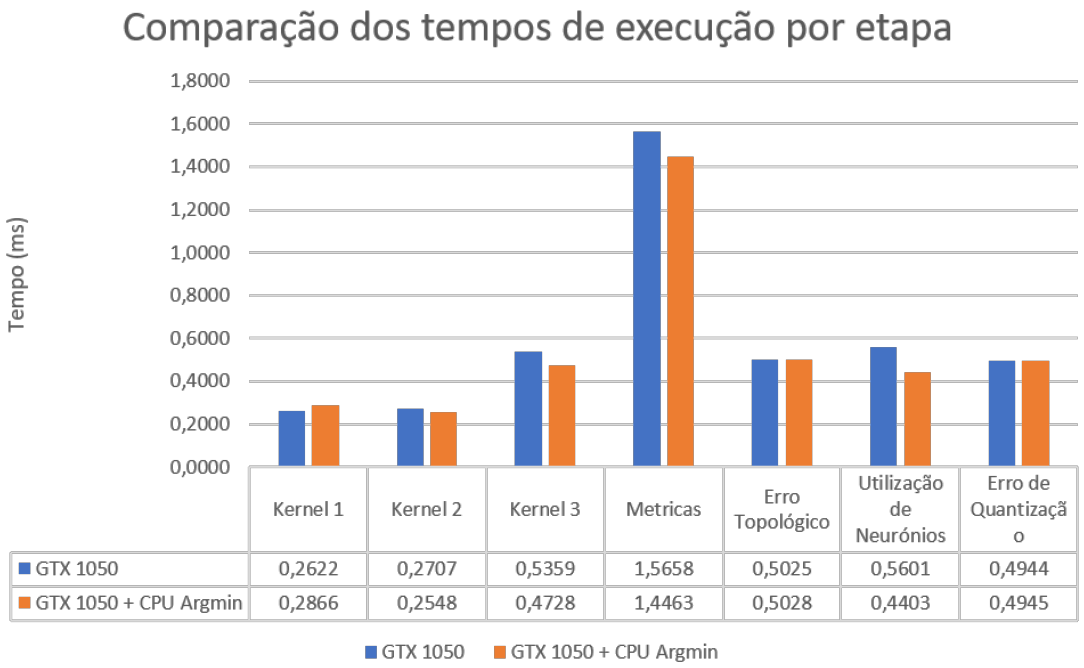


Figura 4.27: Gráfico com o tempo de execução médio de cada etapa com o *dataset* Irís de 16 características, com as operação de *argmin* efetuadas no CPU.

4.5 Impacto do *Mini-Batch*

Também foi analisado o impacto do uso de *mini-batches* na correcção do algoritmo. Esse teste foi efectuado para cada um dos conjuntos de dados.

4.5.1 Íris

Quando comparamos as figuras 4.28 e 4.29, onde se observa as *u-matrixes* e as matrizes de contagem geradas pelo UbiSOM com *batch-sizes* de 100 e 500, com a figura 4.1 (com *batch-size* de 1), é possível observar as diferenças nos *clusters* gerados pelo algoritmo com as diferentes parametrizações. Esta diferença é atribuída ao facto de se recalcular as métricas do UbiSOM entre cada *batch*. Se observarmos os gráficos nas figuras 4.30, 4.31 e 4.32, vemos as ligeiras diferenças na progressão das métricas ao longo da execução do algoritmo.

4.5.2 Chain

As figuras 4.33 e 4.34 mostram o resultado da execução do *dataset* Chain com *batch-size* de 100 e 500. Quando comparado com a figura 4.4 (com *batch-size* de 1), é possível ver que apesar de ligeiras diferenças, os resultados são muito semelhantes, abrindo a possibilidade de com esta *dataset* abdicar de um processamento constante das métricas do UbiSOM, a favor de um ritmo de processamento superior.

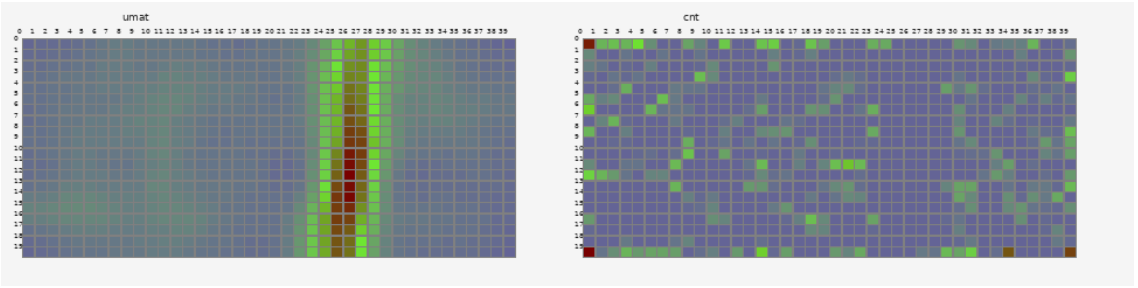


Figura 4.28: *U-Matrix* e Matriz de contagens do UbiSOM com *batch-size* = 100

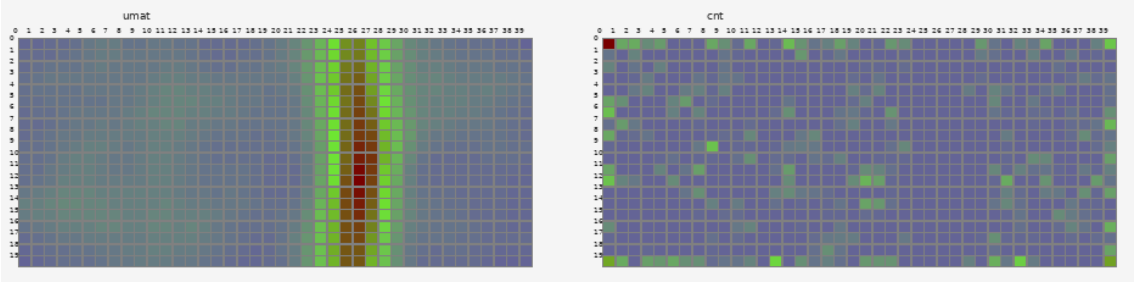


Figura 4.29: *U-Matrix* e Matriz de contagens do UbiSOM com *batch-size* = 500

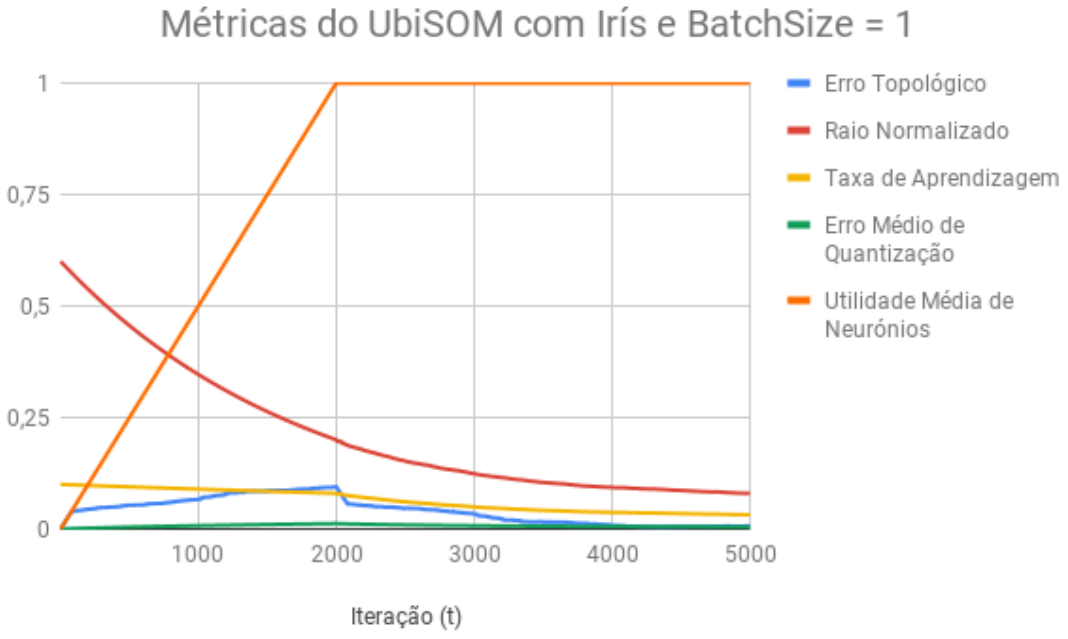


Figura 4.30: Métricas do UbiSOM com *batch-size* = 1

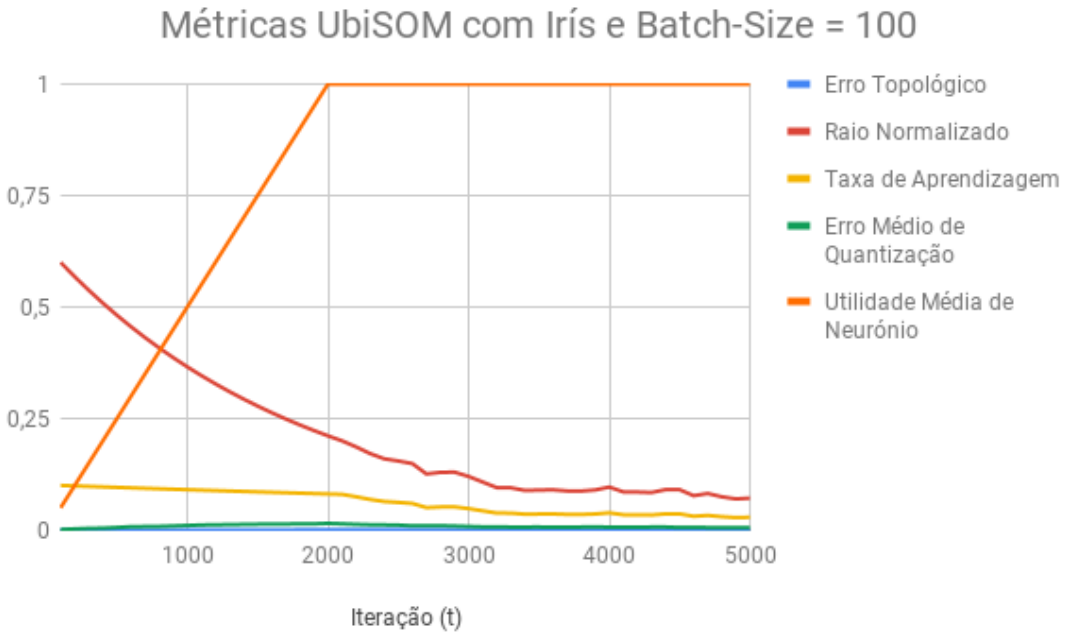


Figura 4.31: Métricas do UbiSOM com *batch-size* = 100

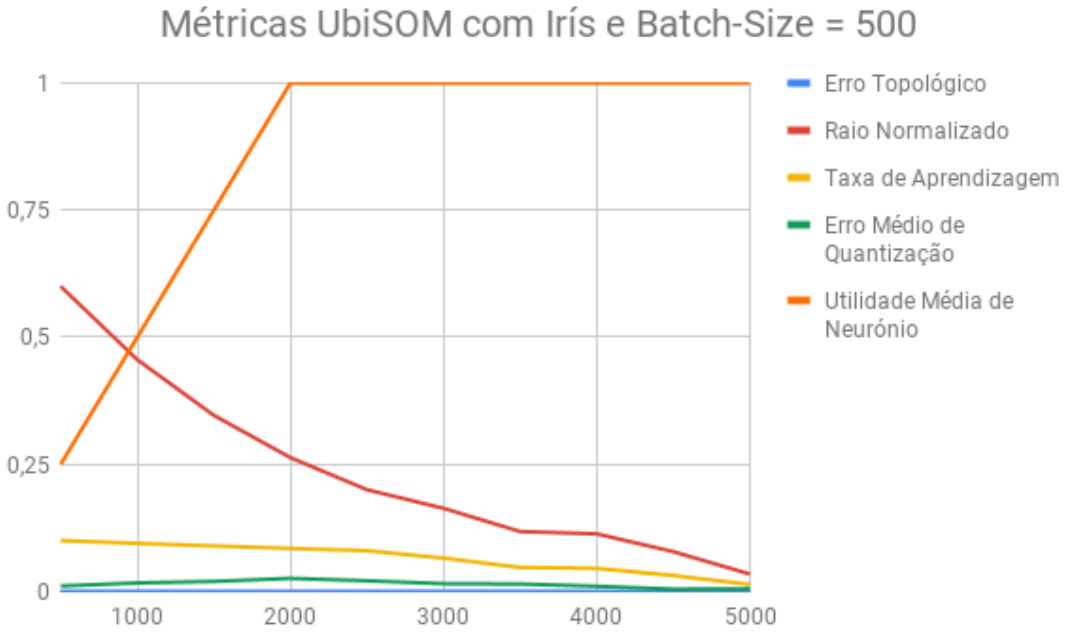


Figura 4.32: Métricas do UbiSOM com *batch-size* = 500

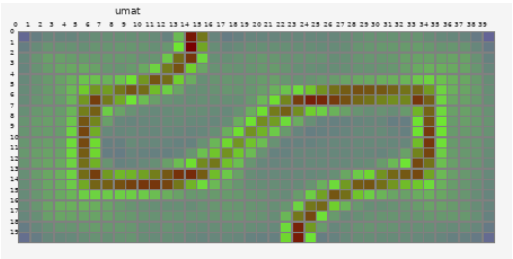


Figura 4.33: Resultado do *dataset* Chain no UbiSOM-Marrow com *batch-size* = 100

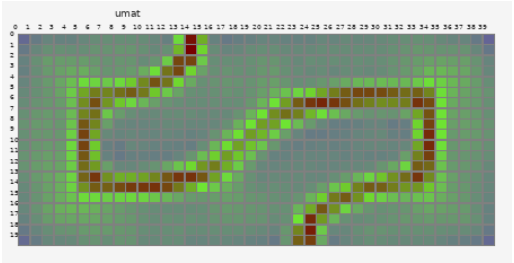


Figura 4.34: Resultado do *dataset* Chain no UbiSOM-Marrow com *batch-size* = 500

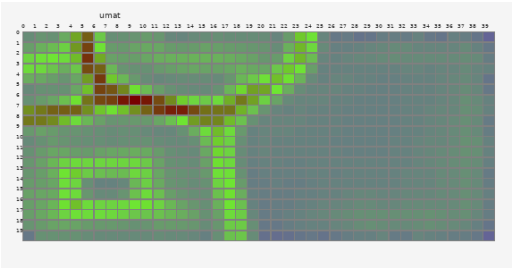


Figura 4.35: Resultado do *dataset* Complex no UbiSOM-Marrow com *batch-size* = 100

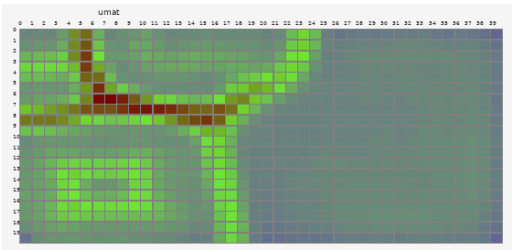


Figura 4.36: Resultado do *dataset* Complex no UbiSOM-Marrow com *batch-size* = 500

4.5.3 Complex

As figuras 4.35 e 4.36 mostram o resultado da execução do *dataset* Complex com *batch-size* de 100 e 500. Quando comparado com os resultados na figura 4.6 (com um *batch-size* de 1) é possível notar que, apesar de ligeiras diferenças, os resultados são muito semelhantes, abrindo a possibilidade de com esta *dataset* abdicar de um processamento constante das métricas do UbiSOM, a favor de um ritmo de processamento superior.

4.5.4 Hepta

As figuras 4.37 e 4.38 mostram o resultado da execução do *dataset* Hepta com *batch-size* de 100 e 500. É interessante notar que no caso do $batch - size = 100$, os resultados foram mais semelhantes em relação aos resultados do UbiSOM original. O modelo com $batch - size = 500$, encontra-se num estado intermédio entre o estado de $batch - size = 1$ e $batch - size = 100$.

4.5.5 Clouds

As figuras 4.39 e 4.40 mostram o resultado da execução do *dataset* Clouds com *batch-size* de 100 e 500. É interessante notar que no caso do $batch - size = 100$, os resultados foram semelhantes em relação aos resultados do UbiSOM original. O modelo com $batch - size =$

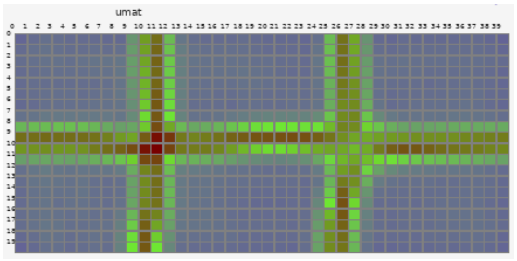


Figura 4.37: Resultado do *dataset* Hepta no UbiSOM-Marrow com *batch-size* = 100

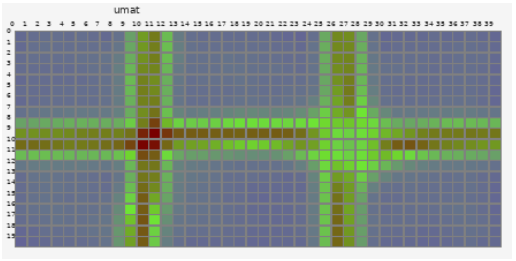


Figura 4.38: Resultado do *dataset* Hepta no UbiSOM-Marrow com *batch-size* = 500

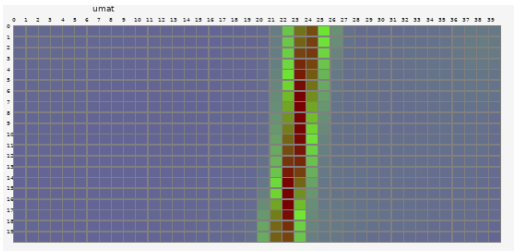


Figura 4.39: Resultado do *dataset* Clouds no UbiSOM-Marrow com *batch-size* = 100

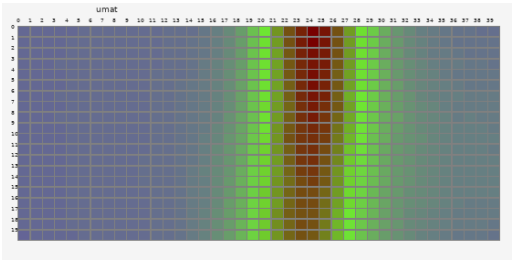


Figura 4.40: Resultado do *dataset* Clouds no UbiSOM-Marrow com *batch-size* = 500

500, por sua vez é semelhante ao $batch - size = 100$, com uma maioria das unidades mais próximas umas das outras.

4.6 Conclusões

Com os testes aqui mencionados é possível concluir vários aspetos em relação à implementação aqui descrita.

Com a análise do *dataset* Irís e das suas variantes com mais características, é possível concluir que o uso de GPUs beneficia imenso o *throughput* do algoritmo UbiSOM, tendo até ganhos de performance de até 77 vezes no Irís com 16 características com a configuração GTX 1050. Também foi possível concluir que não vale a pena executar misturar partes do algoritmo em CPU com partes do algoritmo em GPU, uma vez que qualquer vantagem seria anulada pelos tempos de transferência de dados entre o CPU e o GPU. Também podemos chegar à conclusão de que, mesmo com os ganhos observados pelo uso de um GPU, o CPU continua a ter um efeito bastante forte na velocidade de processamento. Se o CPU não for capaz de acompanhar o ritmo da execução, vai deixar o GPU muitas vezes á espera, reduzindo assim os ganhos de performance.

Com o uso de *datasets* estacionários, como por exemplo o Complex e o Chain, é possível observar que o algoritmo aqui implementado é algoritmicamente equivalente ao UbiSOM e que não existe diferenças significativas á medida que se aumenta o tamanho dos *batches* de observações.

Com o uso de *datasets* não estacionários, encontrou-se diferenças entre as duas versões do algoritmo. Estas diferenças podem ser atribuídas às modificações feitas ao algoritmo para maximizar o seu potencial para uso no GPU. Um ponto inicial para a investigação tal acontecimento aparenta ser a funcionalidade de *reset* do UbiSOM ou possivelmente algum detalhe nalguma das equações de ajuste dinâmico do modelo. Outra justificação é o facto de se terem usados estados iniciais diferentes. Mais investigação sobre este assunto é necessária.

Capítulo 5

Conclusões e Trabalhos Futuros

O objetivo primário desta dissertação foi acelerar a execução do algoritmo UbiSOM com o uso de GPUs. Na secção 4.4.1 podemos ver os resultados desse objetivo. No melhor caso foi possível obter *speedups* de 77 vezes em relação à execução do CPU. Mesmo assim, há que destacar, como observado no gráfico da figura 4.18 que têm-se um aproveitamento bastante limitado do GPU. No entanto é possível observar que com um aumento da dimensionalidade das unidades (dependente do problema em questão), obtemos um *speedup* maior.

Os testes feitos a esta implementação em GPU ainda não replicam exatamente os resultado obtidos com o UbiSOM quando se analisa sua elasticidade perante fontes de dados não estacionarias (secção 4). No entanto os testes efectuados a esta implementação mantêm as fortes capacidades de *clustering*, o que combinado com o aumento da velocidade de processamento, permite um *throughput* maior quando comparado com a sua execução em CPU. Até ao presente momento ainda não foi possível identificar o motivo exato porque a nova versão do algoritmo divergiu da versão original. Uma possível teoria para esta divergência está relacionada com a implementação da funcionalidade de reinício do algorimo ou nalgum detalhe das equações de ajuste dinâmico do modelo. Outra teoria refere ao facto de os estados iniciais do mapa terem sido distintos.

Também foi mostrado na secção 4.5 que em regra, o uso de *mini-batches* tem efeitos mínimos na convergência do mapa nas situações em que esta ocorre de forma semelhante ao UbiSOM original, o que motiva a implementação de otimizações relacionadas com os *mini-batches*, por exemplo, implementar o *mini-batch* como uma matriz, de forma a acelerar ainda mais a execução do algoritmo.

Outro aspeto a considerar foi a utilização da biblioteca Marrow. A motivação do Marrow é fornecer um modelo de programação de GPUs com um foco maior na manipulação

de estruturas de dados, algo que está muito saliente na sua arquitetura, algo que é possível ver nas listagens da secção 3.2. A vasta maioria do que foi implementado consiste em manipulação de coleções de dados, sejam operações aritméticas entre as estruturas, operações de redução ou operações condicionais.

No entanto é preciso ter cuidado com alguns aspetos. Um exemplo está presente na listagem 3.4. Se na linha 8 dessa listagem, a declaração de tipo da variável *neighborhood* fosse definida como **auto**, o Marrow, em vez de executar o *kernel* gerado nas linhas anteriores e armazenar os resultados em *neighborhood*, iria continuar a construir uma AST que só seria executada na linha 12. Por outro lado, iria forçar o recálculo da função de vizinhança ao chegar á operação condicional na linha 18.

Outro aspeto a ter em consideração é o número de transferências que ocorrem entre o CPU e o GPU. Se observarmos as listagens e os diagramas de transferência de memória na secção 3.2.4, relacionada com o cálculo do erro topológico, nomeadamente com a obtenção da 2ª BMU, é preciso transferir coleções inteiras entre o CPU e o GPU sempre que tais operações são executadas. Isto acontece porque existe um valor que é consistentemente alterado nas coleções utilizadas para o cálculo das médias de cada tipo de erro. Por causa desse fator foi tomada a decisão de apenas calcular essas métricas ocasionalmente (entre cada um dos *mini-batches*).

Para além do mencionado, pode ser difícil, numa fase inicial, para o programador adaptar o seu método de raciocínio para o uso de operações coletivas de dados, algo de que o Marrow faz um uso bastante intenso. A falta de documentação formal (neste momento a documentação limita-se a dissertações relacionadas com o Marrow e os vários testes disponibilizados no código fonte) dificulta também a vida do programador que procura implementar um projeto com esta ferramenta.

Sendo uma plataforma ainda em fase de desenvolvimento, existem vários problemas inerentes a esta situação. Em primeiro lugar, como a produção da ferramenta é efetuada em OS X, vários detalhes de implementação, quer a nível de CPU, quer a nível de GPU, podem não transmitir corretamente para outras plataformas. Um exemplo foi a necessidade de melhorar alguns comportamentos do Marrow (efetuadas pelo professor responsável pelo projeto) ao lidar com GPUs da NVIDIA. Outro problema, uma falha que também ficou resolvida pelo professor responsável pelo projeto, foi uma fuga de memoria no GPU que impediu de executar testes de desempenho significativos aos datasets maiores (com 100000 observações ou mais). Após a resolução desse problema, em Linux e com GPUs da NVIDIA (os descritas na secção 4.2.3), ocorre uma falha de segmentação após a execução de alguns *kernels*, relacionado com o facto da ferramenta Marrow não se ter mostrado resiliente o suficiente para lidar com o grande volume de dados, produzido a grande velocidade e a ser processado por múltiplos *threads*.

Há ainda que considerar a implicação do trabalho aqui efetuado. A secção 4.4, mostra-nos que ao aumentar o número de processadores em CPU, os ganhos obtidos são limitados, obtendo um retorno cada vez menor a medida que se adiciona mais *threads*. No entanto,

o uso de GPUs permitiu obter *speedups* muito superiores. Isto implica que, mesmo operações [1] de analises de dados com poucos fundos, podem obter um *throughput* de dados bastante superior através da aquisição de uma placa gráfica de consumidor de baixa gama, como, por exemplo uma NVIDIA GTX 1050 (a 118€ no Newegg[2])[33] ou uma AMD RX 550 (a 78€ no Newegg[2])[35], e obter um ganho superior a um processador de alta gama para consumidores, como um Intel Core i7-8700K (a 330€ no Newegg[2])[34]. Por outro lado, para operações com muitos mais recursos, com as modificações certas ao algoritmo, é possível processar centenas de milhares de observações por segundo e com problemas com centenas de características.

Há que também notar, que apesar do ganho de performance obtido pelo uso de GPU, o CPU continua a ter um efeito bastante pertinente. Se o CPU não tiver capacidade para alimentar o GPU a um ritmo adequado, o GPU vai estar assim mais limitado, e os ganhos de desempenho não serão tão acentuados. Mesmo com GPUs de baixa gama, pode-se observar que em nenhuma ocasião houve uma taxa de ocupação significativa dos GPUs.

5.1 Trabalho futuro

Tendo estas considerações em atenção, há vários tipos de trabalho futuro que se podem propor.

Em relação á implementação do UbiSOM, foi mencionado a divergência entre alguns dos resultados esperados. Reconhecer qual a fonte dessa divergência e identificar uma solução para o problema em questão é uma solução possível para o problema em mãos, levando a procurar uma adaptação do UbiSOM para GPUs que resolva essa divergência.

Existe ainda também a potencialidade de utilizar mapas UbiSOM como funções de selecção de BMU, funções de vizinhanças, funções de cálculos de taxa de aprendizagem, entre outras possíbilidades. Para este efeito, seria também interessante explorar a capacidade de distribuir os diferentes mapas por vários GPUs.

Outra possibilidade é procurar aumentar a proporção de tempo que o algoritmo gasta no GPU em comparação com tempo que gasta no CPU e a transferir informações entre os dois (como mostrado pela baixa taxa de utilização em ambos os GPUs no gráfico 4.18). Uma possível acção é transferir de imediato todas as observações a ser processadas para dentro do GPU, evitando ter que transferir cada observação individualmente à medida que essa observação é necessária. Outra melhoria possível de implementar é a capacidade de selecionar o i-ésimo menor/maior elemento de uma coleção, para evitar situações como a presente na secção 3.2.4, em que é preciso retornar a coleção de dados ao CPU e depois enviá-la de novo para o GPU por causa da modificação de um valor.

Por fim, a melhoria continua do Marrow, através da adição de novos *skeletons* para efectuar operações típicas como por exemplo o *Stencyl* (teria sido útil para a implementação do cálculo da *u-matrix* em GPU), reparar alguns dos problemas atualmente existentes no

[1] PMEs, p.e..
[2] No momento em que esta dissertação foi escrita.

Marrow, ou adicionar suporte para sistemas operativos móveis (por exemplo, com suporte ao OpenCL ES) são trabalhos que podem ser efetuados no futuro.

Bibliografia

[1] I. Advanced Micro Devices. *AMD Accelerated Parallel Processing OpenCL Programming Guide*. 2013. URL: http://developer.amd.com/wordpress/media/2013/07/AMD_Accelerated_Parallel_Processing_OpenCL_Programming_Guide-rev-2.7.pdf.

[2] A. Akinduko, E. Mirkes e A Gorban. "SOM: Stochastic initialization versus principal components". Em: 364–365 (out. de 2016), pp. 213–221.

[3] E. Alerstam, T. Svensson e S. Andersson-Engels. "Parallel computing with graphics processing units for high-speed Monte Carlo simulation of photon migration". Em: *Journal of Biomedical Optics* 13.6 (2008). DOI: 10.1117/1.3041496.

[4] P. Antonellis, C. Makris e N. Tsirakis. "Algorithms for Clustering Clickstream Data". Em: *Inf. Process. Lett.* 109.8 (mar. de 2009), pp. 381–385. ISSN: 0020-0190. DOI: 10.1016/j.ipl.2008.12.011. URL: http://dx.doi.org/10.1016/j.ipl.2008.12.011.

[5] D. Barbará. "Requirements for Clustering Data Streams". Em: *SIGKDD Explor. Newsl.* 3.2 (jan. de 2002), pp. 23–27. ISSN: 1931-0145. DOI: 10.1145/507515.507519. URL: http://doi.acm.org/10.1145/507515.507519.

[6] J. Behrens. "Principles and Procedures of Exploratory Data Analysis". Em: 2 (jun. de 1997), pp. 131–160.

[7] H. Benitez-Perez, J. L. e A. Benitez. "Using Wavelets for Feature Extraction and Self Organizing Maps for Fault Diagnosis of Nonlinear Dynamic Systems". Em: *Applications of Self-Organizing Maps* (2012). DOI: 10.5772/50235.

[8] K. Boydstun. *Introduction to OpenCL*. Ago. de 2011. URL: http://www.tapir.caltech.edu/~kboyds/OpenCL/opencl.pdf.

[9] T. Cavazos. "Using Self-Organizing Maps to Investigate Extreme Climate Events: An Application to Wintertime Precipitation in the Balkans." Em: *Journal of Climate* 13.10 (2000), p. 1718. ISSN: 08948755. URL: http://widgets.ebscohost.com/prod/customerspecific/ns000290/authentication/index.php?url=http%3a%2f%2fsearch.ebscohost.com%2flogin.aspx%3fdirect%3dtrue%26AuthType%3dip%2ccookie%2cshib%2cuid%26db%3da9h%26AN%3d5594311%26lang%3dpt-br%26site%3deds-live%26scope%3dsite.

[10] *CUDA LLVM Compiler*. 2017. URL: https://developer.nvidia.com/cuda-llvm-compiler.

[11] A. De, Y. Zhang e C. Guo. "A parallel adaptive segmentation method based on SOM and GPU with application to MRI image processing". Em: *Neurocomputing* 198 (2016). Advances in Neural Networks, Intelligent Control and Information Processing, pp. 180 –189. ISSN: 0925-2312. DOI: https://doi.org/10.1016/j.neucom.2015.10.129. URL: http://www.sciencedirect.com/science/article/pii/S0925231216003283.

[12] J. I. Deichmann, A. Eshghi, D. Haughton e M. Li. "Socioeconomic Convergence in Europe One Decade After the EU Enlargement of 2004: Application of Self-Organizing Maps." Em: *Eastern European Economics* 55.3 (2017), pp. 236 –260. ISSN: 00128775. URL: http://widgets.ebscohost.com/prod/customerspecific/ns000290/authentication/index.php?url=http%3a%2f%2fsearch.ebscohost.com%2flogin.aspx%3fdirect%3dtrue%26AuthType%3dip%2ccookie%2cshib%2cuid%26db%3dbth%26AN%3d123434682%26lang%3dpt-br%26site%3deds-live%26scope%3dsite.

[13] E Erwin, K Obermayer e K Schulten. "Self-organizing maps: ordering, convergence properties and energy functions." Em: *Biological Cybernetics* 67.1 (1992), pp. 47 –55. ISSN: 0340-1200. URL: http://widgets.ebscohost.com/prod/customerspecific/ns000290/authentication/index.php?url=http%3a%2f%2fsearch.ebscohost.com%2flogin.aspx%3fdirect%3dtrue%26AuthType%3dip%2ccookie%2cshib%2cuid%26db%3dmnh%26AN%3d1606243%26lang%3dpt-br%26site%3deds-live%26scope%3dsite.

[14] R. Farivar, D. Rebolledo, E. Chan e R. Campbell. "A parallel implementation of K-means clustering on GPUs". Em: *Proceedings of the 2008 International Conference on Parallel and Distributed Processing Techniques and Applications, PDPTA 2008*. 2008, pp. 340–345. ISBN: 1601320841.

[15] P. Ferrão, H. Marques e H. Paulino. "Stream Processing on Hybrid CPU/Intel® Xeon Phi™, Systems". Em: *Euro-Par 2018: Parallel Processing - 24th International Conference on Parallel and Distributed Computing, Turin, Italy, August 27-31, 2018, Proceedings*. Vol. 11014. Lecture Notes in Computer Science. Springer, 2018, pp. 796–810. DOI: 10.1007/978-3-319-96983-1_56. URL: https://doi.org/10.1007/978-3-319-96983-1_56.

[16] M. Fialho. *From C++ Expressions to GPU Algorithms*. Rel. téc. Universidade Nova de Lisboa, Faculdade de Ciências e Tecnologias.

[17] J.-C. Fort, P. Letrémy e M. Cottrell. "Advantages and drawbacks of the Batch Kohonen algorithm." Em: *European Symposium on Artificial Neural Networks (ESANN)*. Vol. 2. 2002, pp. 223–230.

[18] J. Fung e S. Mann. "Computer vision signal processing on graphics processing units". Em: *2004 IEEE International Conference on Acoustics, Speech, and Signal Processing* (mai. de 2004). DOI: 10.1109/icassp.2004.1327055.

[19] J. Fung, F. Tang e S. Mann. "Mediated reality using computer graphics hardware for computer vision". Em: *Proceedings. Sixth International Symposium on Wearable Computers*, (out. de 2002). DOI: 10.1109/iswc.2002.1167222.

[20] M. Gaber, J. Gama, S. Krishnaswamy, J. Gomes e F. Stahl. "Data stream mining in ubiquitous environments: State-of-the-art and current directions". Em: 4 (mar. de 2014), pp. 116–138.

[21] J. a. Gama, P. P. Rodrigues e L. Lopes. "Clustering Distributed Sensor Data Streams Using Local Processing and Reduced Communication". Em: *Intell. Data Anal.* 15.1 (jan. de 2011), pp. 3–28. ISSN: 1088-467X. URL: http://dl.acm.org/citation.cfm?id=1937721.1937723.

[22] J. Gama. *Knowledge Discovery from Data Streams*. 1st. Chapman & Hall/CRC, 2010. ISBN: 1439826110, 9781439826119.

[23] *Graphics Processing Unit (GPU)*. Accessed on 19.11.2017. URL: http://www.nvidia.com/object/gpu.html.

[24] K. O. W. Group. *The OpenCL Specification*. Mai. de 2017. URL: https://www.khronos.org/registry/OpenCL/specs/opencl-2.2.html#_introduction.

[25] G. Hamerly e C. Elkan. "Alternatives to the k-means algorithm that find better clusterings". Em: *Proceedings of the eleventh international conference on Information and knowledge management - CIKM 02* (2002). DOI: 10.1145/584887.584890.

[26] O. Harrison e J. Waldron. "AES Encryption Implementation and Analysis on Commodity Graphics Processing Units". Em: *9th Workshop on Cryptographic Hardware and Embedded Systems (CHES 2007*, pp. 209–226.

[27] O. Harrison e J. Waldron. "Practical Symmetric Key Cryptography on Modern Graphics Hardware". Em: (2008).

[28] O. Harrison e J. Waldron. *Efficient Acceleration of Asymmetric Cryptography on Graphics Hardware*. 2009.

[29] J. A. Hartigan e M. A. Wong. "A k-means clustering algorithm". Em: *JSTOR: Applied Statistics* 28.1 (1979), pp. 100–108.

[30] K. S. Hasan, A. Chatterjee, S. Radhakrishnan e J. K. Antonio. "Performance Prediction Model and Analysis for Compute-Intensive Tasks on GPUs". Em: *11th IFIP International Conference on Network and Parallel Computing (NPC)*. Ed. por C.-H. Hsu, X. Shi e V. Salapura. Vol. LNCS-8707. Network and Parallel Computing. Part 6: Poster Sessions. Ilan, Taiwan: Springer, set. de 2014, pp. 612–617. DOI: 10.1007/978-3-662-44917-2_65. URL: https://hal.inria.fr/hal-01403164.

[31] *Heterogeneous Processing: a Strategy for Augmenting Moore's Law*. 2006. URL: http://www.linuxjournal.com/article/8368.

[32] F. R. A. Hopgood, R. J. Hubbold e D. A. Duce. *Advances in computer graphics II*. Springer, 1986.

[33] N. I. Inc. *EVGA GeForce GTX 1050 GAMING, 02G-P4-6150-KR, 2GB GDDR5, DX12 OSD Support (PXOC)*. 2017. URL: https://www.newegg.com/global/pt/Product/Product.aspx?Item=N82E16814487295&cm_re=GTX_1050-_-14-487-295-_-Product.

[34] N. I. Inc. *Intel Core i7-8700K Coffee Lake 6-Core 3.7 GHz (4.7 GHz Turbo) LGA 1151 (300 Series) 95W BX80684I78700K Desktop Processor Intel UHD Graphics 630*. 2017. URL: https://www.newegg.com/global/pt/Product/Product.aspx?Item=N82E16819117827.

[35] N. I. Inc. *PowerColor Radeon RX 550 DirectX 12 AXRX 550 2GBD5-DHA/OC 2GB 128-Bit GDDR5 PCI Express 3.0 CrossFireX Support ATX Video Card*. 2017. URL: https://www.newegg.com/global/pt/Product/Product.aspx?Item=N82E16814131738&cm_re=RX_550-_-14-131-738-_-Product.

[36] Y. Jia, E. Shelhamer, J. Donahue, S. Karayev, J. Long, R. Girshick, S. Guadarrama e T. Darrell. "Caffe: Convolutional Architecture for Fast Feature Embedding". Em: *arXiv preprint arXiv:1408.5093* (2014).

[37] M. Johnsson. "Aplications Of Self-Organizing Maps". Em: (2012). URL: http://www.novaims.unl.pt/docentes/vlobo/Publicacoes/4_1_03_Applications_of_Self-Organizing_Maps.pdf.

[38] H. Kargupta, B.-H. Park, S. Pittie, L. Liu, D. Kushraj e K. Sarkar. "MobiMine: Monitoring the Stock Market from a PDA". Em: *ACM SIGKDD Explorations* 3 (2001), pp. 37–46.

[39] H. Kargupta, R. Bhargava, K. Liu, M. Powers, P. Blair, S. Bushra, J. Dull, K. Sarkar, M. Klein, M. Vasa e D. Handy. "VEDAS: A Mobile and Distributed Data Stream Mining System for Real-Time Vehicle Monitoring". Em: (abr. de 2004).

[40] T. Kohonen, M. R. Schroeder e T. S. Huang, eds. *Self-Organizing Maps*. 3rd. Secaucus, NJ, USA: Springer-Verlag New York, Inc., 2001. ISBN: 3540679219.

[41] T. Kohonen e T. Honkela. *Kohonen network*. Jan. de 2007. URL: http://www.scholarpedia.org/article/Kohonen_network.

[42] M. Kontaki, A. N. Papadopoulos e Y. Manolopoulos. "Continuous Trend-Based Clustering in Data Streams". Em: *Proceedings of the 10th International Conference on Data Warehousing and Knowledge Discovery*. DaWaK '08. Turin, Italy: Springer-Verlag, 2008, pp. 251–262. ISBN: 978-3-540-85835-5. DOI: 10.1007/978-3-540-85836-2_24. URL: http://dx.doi.org/10.1007/978-3-540-85836-2_24.

[43] T. Kuremoto, T. Komoto, K. Kobayashi e M. Obayashi. "Parameterless-Growing-SOM and Its Application to a Voice Instruction Learning System". Em: *Journal of Robotics* 2010 (2010), 1–9. DOI: 10.1155/2010/307293.

[44] S. Lloyd. "Least squares quantization in PCM". Em: *IEEE Transactions on Information Theory* 28.2 (1982), pp. 129–137. ISSN: 0018-9448. DOI: 10.1109/TIT.1982.1056489.

[45] Z. P. Lo e B. Bavarian. "On the rate of convergence in topology preserving neural networks". Em: *Biological Cybernetics* 65.1 (mai. de 1991), pp. 55–63. ISSN: 1432-0770. DOI: 10.1007/BF00197290. URL: https://doi.org/10.1007/BF00197290.

[46] *Machine Learning Applications for High Performance Computing - NVIDIA*. URL: http://www.nvidia.com/object/machine-learning.html.

[47] *Machine Learning Applications for High Performance Computing - NVIDIA*. URL: http://www.nvidia.com/object/machine-learning.html.

[48] D. J. C. MacKay. *Information theory, inference, and learning algorithms*. Cambridge University Press, 2017.

[49] S. A. Manavski e G. Valle. "CUDA compatible GPU cards as efficient hardware accelerators for Smith-Waterman sequence alignment". Em: *BMC Bioinformatics* 9.Suppl 2 (2008). DOI: 10.1186/1471-2105-9-s2-s10.

[50] N. Mancheril. "GPU-based Sorting in PostgreSQL". Em: ().

[51] R. Marques, H. Paulino, F. Alexandre e P. D. Medeiros. "Algorithmic Skeleton Framework for the Orchestration of GPU Computations". Em: *Euro-Par 2013 Parallel Processing*. Ed. por F. Wolf, B. Mohr e D. an Mey. Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, pp. 874–885. ISBN: 978-3-642-40047-6.

[52] S. Mathew e P. Joy. "Ultra Fast SOM using CUDA". Em: (dez. de 2010).

[53] S. Mcconnell, R. Sturgeon, G. Henry, A. Mayne e R. Hurley. "Scalability of self-organizing maps on a GPU cluster using OpenCL and CUDA". Em: 341 (fev. de 2012), p. 012018.

[54] J. C.C.B.S. D. Mello, E. Goncalves, L. Angulo, L. Biondi, U. G.P. D. Abreu, T. B. D. Carvalho e S. De. "Ex-Post Clustering of Brazilian Beef Cattle Farms Using Soms and Cross-Evaluation Dea Models". Em: *Applications of Self-Organizing Maps* (2012). DOI: 10.5772/51324.

[55] F. C. Moraes, S. C. Botelho, N. D. Filho e J. F. O. Gaya. "Parallel High Dimensional Self Organizing Maps Using CUDA". Em: *2012 Brazilian Robotics Symposium and Latin American Robotics Symposium*. 2012, pp. 302–306. DOI: 10.1109/SBR-LARS.2012.56.

[56] W Natita, W Wiboonsak e S Dusadee. "Appropriate Learning Rate and Neighborhood Function of Self-organizing Map (SOM) for Specific Humidity Pattern Classification over Southern Thailand". Em: 6 (jan. de 2016), pp. 61–65.

[57] O. Omitaomu, R. Vatsavai, A. Ganguly, N. Chawla, J. Gama e M. Gaber. *Knowledge discovery from sensor data (SensorKDD)*. Vol. 11. Jan. de 2009, pp. 84–87.

[58] *OpenCL - The open standard for parallel programming of heterogeneous systems*. 2013. URL: https://www.khronos.org/opencl/.

[59] *OpenCL - The open standard for parallel programming of heterogeneous systems*. 2013. URL: https://www.khronos.org/opencl/resources.

[60] *OpenCL Gains Ground On CUDA*. 2014. URL: https://www.hpcwire.com/2012/02/28/opencl_gains_ground_on_cuda/.

[61] H. Paulino, D. Parreira, N. Delgado, A. Ravara e A. G. A. Matos. "From atomic variables to data-centric concurrency control". Em: *Proceedings of the 31st Annual ACM Symposium on Applied Computing, Pisa, Italy, April 4-8, 2016*. Ed. por S. Ossowski. ACM, 2016, pp. 1806–1811. DOI: 10.1145/2851613.2851734. URL: https://doi.org/10.1145/2851613.2851734.

[62] *Protocol Buffers | Google Developers*. URL: https://developers.google.com/protocol-buffers/.

[63] M. Resta. "Graph Mining Based SOM: A Tool to Analyze Economic Stability". Em: *Applications of Self-Organizing Maps* (2012). DOI: 10.5772/51240.

[64] P. P. Rodrigues, J. a. Gama e L. Lopes. "Clustering Distributed Sensor Data Streams". Em: *Proceedings of the European Conference on Machine Learning and Knowledge Discovery in Databases - Part II*. ECML PKDD '08. Antwerp, Belgium: Springer-Verlag, 2008, pp. 282–297. ISBN: 978-3-540-87480-5. DOI: 10.1007/978-3-540-87481-2_19. URL: http://dx.doi.org/10.1007/978-3-540-87481-2_19.

[65] M. C. Schatz, C. Trapnell, A. L. Delcher e A. Varshney. "High-throughput sequence alignment using Graphics Processing Units". Em: *BMC Bioinformatics* 8.1 (2007). DOI: 10.1186/1471-2105-8-474.

[66] S. C. Sheridan e C. C. Lee. "The self-organizing map in synoptic climatological research." Em: *Progress in Physical Geography* 35.1 (2011), pp. 109 –119. ISSN: 03091333. URL: http://widgets.ebscohost.com/prod/customerspecific/ns000290/authentication/index.php?url=http%3a%2f%2fsearch.ebscohost.com%2flogin.aspx%3fdirect%3dtrue%26AuthType%3dip%2ccookie%2cshib%2cuid%26db%3da9h%26AN%3d57788536%26lang%3dpt-br%26site%3deds-live%26scope%3dsite.

[67] B. Silva e N. Marques. "Feature Clustering with Self-organizing Maps and an Application to Financial Time-series for Portfolio Selection." Em: (2010). URL: https://www.researchgate.net/publication/221616457_Feature_Clustering_with_Self-organizing_Maps_and_an_Application_to_Financial_Time-series_for_Portfolio_Selection.

[68] B. Silva e N. C. Marques. "The ubiquitous self-organizing map for non-stationary data streams". Em: *Journal of Big Data* 2.1 (2015), p. 27. ISSN: 2196-1115. DOI: 10.1186/s40537-015-0033-0. URL: https://doi.org/10.1186/s40537-015-0033-0.

[69] B. M. N. da Silva. "Exploratory Cluster Analysis from Ubiquitous Data Streams using Self-Organizing Maps". Tese de doutoramento. Faculdade de Ciências e Técnologia da Universidade Nova de Lisboa, 2016.

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

[71] F. Stahl, M. M. Gaber, M. Bramer e P. S. Yu. "Pocket Data Mining: Towards Collaborative Data Mining in Mobile Computing Environments". Em: *2010 22nd IEEE International Conference on Tools with Artificial Intelligence*. Vol. 2. 2010, pp. 323–330. DOI: 10.1109/ICTAI.2010.118.

[72] M. Steuwer, P. Kegel e S. Gorlatch. "SkelCL - A Portable Skeleton Library for High-Level GPU Programming". Em: *Proceedings of the 2011 IEEE International Symposium on Parallel and Distributed Processing Workshops and PhD Forum*. IPDPSW '11. Washington, DC, USA: IEEE Computer Society, 2011, pp. 1176–1182. ISBN: 978-0-7695-4577-6. DOI: 10.1109/IPDPS.2011.269. URL: http://dx.doi.org/10.1109/IPDPS.2011.269.

[73] J. W. Tukey. "The Future of Data Analysis". Em: *Ann. Math. Statist.* 33.1 (mar. de 1962), pp. 1–67. DOI: 10.1214/aoms/1177704711. URL: https://doi.org/10.1214/aoms/1177704711.

[74] *UCI Machine Learning Repository: Iris Data Set*. URL: https://archive.ics.uci.edu/ml/datasets/Iris/.

[75] G. Vasiliadis e S. Ioannidis. "GrAVity: A Massively Parallel Antivirus Engine". Em: *Lecture Notes in Computer Science Recent Advances in Intrusion Detection* (2010). DOI: 10.1007/978-3-642-15512-3_5.

[76] B. Wilson. *The Machine Learning Dictionary*. http://www.cse.unsw.edu.au/~billw/dictionaries/mldict.html.

[77] P. Wittek e S. Daranyi. "A GPU-Accelerated Algorithm for Self-Organizing Maps in a Distributed Environment". Em: (abr. de 2012).

[78] P. Wittek, S. Gao, I. Lim e L. Zhao. "somoclu: An Efficient Parallel Library for Self-Organizing Maps". Em: *Journal of Statistical Software, Articles* 78.9 (2017), pp. 1–21. ISSN: 1548-7660. DOI: 10.18637/jss.v078.i09. URL: https://www.jstatsoft.org/v078/i09.

[79] M. Zechner e M. Granitzer. "Accelerating K-Means on the Graphics Processor via CUDA". Em: *2009 First International Conference on Intensive Applications and Services*. 2009, pp. 7–15. DOI: 10.1109/INTENSIVE.2009.19.

[80] X. Zhang e W. Wang. "Self-adaptive Change Detection in Streaming Data with Non-stationary Distribution". Em: *Advanced Data Mining and Applications*. Ed. por L. Cao, Y. Feng e J. Zhong. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, pp. 334–345. ISBN: 978-3-642-17316-5.

Apêndice 1 *Kernels* OpenCL gerados pelo Marrow

Neste apêndice estarão visíveis os *kernels* OpenCL gerados pelo Marrow para a execução do algoritmo implementado. Nenhum destes *kernels* foi implementado pelo autor desta dissertação.

Listagem A.1: 1º *Kernel* OpenCL do Cálculo de Distância.

```
__kernel void marrow_kernel(global float* out_matrix0,
  global float* matrix1,
  global float* array2,
  const unsigned long limit0,
  const unsigned long limit1)
{
const unsigned long index0 = get_global_id(0);
if (index0 >= limit0) return;
const unsigned long index1= get_global_id(1);
if (index1 >= limit1) return;
const unsigned long index01 =  index1 * limit0 + index0;
float local0 = matrix1[index01] - array2[index0];
out_matrix0[index01] = local0 * local0;
}
```

Listagem A.2: *Kernel* OpenCL para obter o índice do valor mínimo de um *array*.

```
#define T float
#define Operation(X, Y) (min (X, Y))

__kernel void marrow_kernel(__global T *g_idata,
  __global T *g_odata,
```

```
6       __global unsigned *indexes,
7       const unsigned long n,
8       __local volatile T* sdata,
9       __local volatile unsigned long* idata)
10  {
11
12      size_t blockSize = get_local_size(0);
13      size_t tid = get_local_id(0);
14      size_t first = get_group_id(0) * 2 * blockSize + tid;
15      size_t gridSize = blockSize * 2 * get_num_groups(0);
16      T current;
17
18      size_t i = first;
19      if (i < n) {
20          current = sdata[tid] = g_idata[i];
21          idata[tid] = i;
22
23          if (i + blockSize < n) {
24              current = Operation(sdata[tid], g_idata[i + blockSize]);
25              if (current != sdata[tid]) {
26                  sdata[tid] = current;
27                  idata[tid] = i + blockSize;
28              }
29          }
30          i += gridSize;
31
32          while (i < n) {
33            current = Operation(sdata[tid], g_idata[i]);
34            if (current != sdata[tid]) {
35                sdata[tid] = current;
36                idata[tid] = i;
37            }
38
39            if (i + blockSize < n) {
40              current = Operation(sdata[tid], g_idata[i + blockSize]);
41              if (current != sdata[tid]) {
42                  sdata[tid] = current;
43                  idata[tid] = i + blockSize;
44              }
45            }
46            i += gridSize;
47          }
48      }
49
50      barrier(CLK_LOCAL_MEM_FENCE);
51
```

```
52      if (tid < 128 && (first + 128 < n)) {
53        current = Operation(sdata[tid], sdata[tid + 128]);
54        if (current != sdata[tid]) {
55          sdata[tid] = current;
56          idata[tid] = idata[tid + 128];
57        }
58      }
59      barrier(CLK_LOCAL_MEM_FENCE);
60
61      if (tid < 64 && (first + 64 < n)) {
62        current = Operation(sdata[tid], sdata[tid + 64]);
63        if (current != sdata[tid]) {
64          sdata[tid] = current;
65          idata[tid] = idata[tid + 64];
66        }
67      }
68      barrier(CLK_LOCAL_MEM_FENCE);
69
70      if (tid < 32) {
71        if (first + 32 < n) {
72          current = Operation(sdata[tid], sdata[tid + 32]);
73          if (current != sdata[tid]) {
74            sdata[tid] = current;
75            idata[tid] = idata[tid + 32];
76          }
77        }
78        if (first + 16 < n) {
79          current = Operation(sdata[tid], sdata[tid + 16]);
80          if (current != sdata[tid]) {
81            sdata[tid] = current;
82            idata[tid] = idata[tid + 16];
83          }
84        }
85        if (first +  8 < n) {
86          current = Operation(sdata[tid], sdata[tid + 8]);
87          if (current != sdata[tid]) {
88           sdata[tid] = current;
89           idata[tid] = idata[tid + 8];
90          }
91        }
92       if (first +  4 < n) {
93          current = Operation(sdata[tid], sdata[tid + 4]);
94          if (current != sdata[tid]) {
95           sdata[tid] = current;
96           idata[tid] = idata[tid + 4];
97          }
```

```
98      }
99      if (first + 2 < n) {
100       current = Operation(sdata[tid], sdata[tid + 2]);
101       if (current != sdata[tid]) {
102         sdata[tid] = current;
103         idata[tid] = idata[tid + 2];
104       }
105     }
106     if (first + 1 < n) {
107       current = Operation(sdata[tid], sdata[tid + 1]);
108       if (current != sdata[tid]) {
109         sdata[tid] = current;
110         idata[tid] = idata[tid + 1];
111       }
112     }
113   }
114
115   if (tid == 0) {
116     g_odata[get_group_id(0)] = sdata[0];
117     indexes[get_group_id(0)] = idata[0];
118   }
119 }
```

Listagem A.3: *Kernel* OpenCL para o cálculo da função de vizinhança.

```
1  __kernel void marrow_kernel(global float* out_array0,
2    global float* array1,
3    const float float_2,
4    const int int_3,
5    const float float_4,
6    const unsigned long limit0)
7  {
8  const size_t index0 = get_global_id(0);
9  if (index0 >= limit0) return;
10 float local0 = array1[index0] / float_2;
11 float local1 = pow(local0, int_3);
12 float local2 = local1 * float_4;
13 out_array0[index0] = exp(local2);
14 }
```

Listagem A.4: *Kernel* OpenCL para o cálculo do modificador do método de atualização do mapa.

```
1  __kernel void marrow_kernel(global float* out_array7,
2    global float* array8,
3    const float float_9,
4    const float float_10,
```

```
  const float float_11,
  const unsigned long limit0)
{
const size_t index0 = get_global_id(0);
if (index0 >= limit0) return;
float local2 = array8[index0] > float_9;
float local3 = local2 ? array8[index0] : float_10;
out_array7[index0] = local3 * float_11;
}
```

Listagem A.5: *Kernel* OpenCL para a atualização do mapa.

```
__kernel void marrow_kernel(global float* out_matrix12,
  global float* matrix13,
  global float* array14,
  global float* array15,
  const unsigned long limit0,
  const unsigned long limit1)
{
const size_t index0 = get_global_id(0);
if (index0 >= limit0) return;
const size_t index1= get_global_id(1);
if (index1 >= limit1) return;
const size_t index01 =  index1 * limit0 + index0;
float local4 = array14[index0] - matrix13[index01];
float local5 = local4 * array15[index1];
out_matrix12[index01] = matrix13[index01] + local5;
}
```

Listagem A.6: *Kernel* OpenCL para a atualização de *t_k_update*.

```
__kernel void marrow_kernel(global int* out_array16,
  global float* array17,
  const float float_18,
  const int int_19,
  global int* array20,
  const unsigned long limit0)
{
const size_t index0 = get_global_id(0);
if (index0 >= limit0) return;
float local6 = array17[index0] > float_18;
out_array16[index0] = local6 ? int_19 : array20[index0];
}
```

Listagem A.7: *Kernel* OpenCL para redução de um *array* de floats para um escalar.

```
#define T float
```

```
2  #define Operation(X, Y) ((X) + (Y))
3
4  __kernel void marrow_kernel(__global T *g_idata,
5    __global T *g_odata,
6    const unsigned long n,
7    __local volatile T* sdata)
8  {
9      size_t blockSize = get_local_size(0);
10     size_t tid = get_local_id(0);
11     size_t first = get_group_id(0) * 2 * blockSize + tid;
12     size_t gridSize = blockSize * 2 * get_num_groups(0);
13
14     size_t i = first;
15     if (i < n) {
16        sdata[tid] = g_idata[i];
17
18        if (i + blockSize < n)
19          sdata[tid] = Operation(sdata[tid], g_idata[i + blockSize]);
20        i += gridSize;
21
22        while (i < n) {
23         sdata[tid] = Operation(sdata[tid], g_idata[i]);
24
25          if (i + blockSize < n)
26              sdata[tid] = Operation(sdata[tid],
27               g_idata[i + blockSize]);
28          i += gridSize;
29        }
30     }
31
32     barrier(CLK_LOCAL_MEM_FENCE);
33
34     if (tid < 128 && (first + 128 < n))
35     {
36     sdata[tid] = Operation(sdata[tid], sdata[tid + 128]);
37     }
38     barrier(CLK_LOCAL_MEM_FENCE);
39
40     if (tid < 64 && (first + 64 < n))
41     {
42     sdata[tid] = Operation(sdata[tid], sdata[tid + 64]);
43     }
44     barrier(CLK_LOCAL_MEM_FENCE);
45
46     if (tid < 32) {
47        if (first + 32 < n)
```

```
48              sdata[tid] = Operation(sdata[tid], sdata[tid + 32]);
49          if (first + 16 < n)
50              sdata[tid] = Operation(sdata[tid], sdata[tid + 16]);
51          if (first +  8 < n)
52              sdata[tid] = Operation(sdata[tid], sdata[tid + 8]);
53          if (first +  4 < n)
54              sdata[tid] = Operation(sdata[tid], sdata[tid + 4]);
55          if (first +  2 < n)
56              sdata[tid] = Operation(sdata[tid], sdata[tid + 2]);
57          if (first +  1 < n)
58              sdata[tid] = Operation(sdata[tid], sdata[tid + 1]);
59      }
60
61      if (tid == 0)
62          g_odata[get_group_id(0)] = sdata[0];
63  }
```

Listagem A.8: 1º *Kernel* OpenCL para o cálculo da utilidade de neurónios.

```
1  __kernel void marrow_kernel(global int* out_array21,
2    const int int_22,
3    global int* array23,
4    const int int_24,
5    const int int_25,
6    const int int_26,
7    const unsigned long limit0)
8  {
9  const size_t index0 = get_global_id(0);
10 if (index0 >= limit0) return;
11 int local7 = -array23[index0];
12 int local8 = local7 + int_24;
13 int local9 = int_22 > local8;
14 out_array21[index0] = local9 ? int_25 : int_26;
15 }
```

Listagem A.9: 1º *Kernel* OpenCL para soma de um *array* de inteiros.

```
1  #define T int
2  #define Operation(X, Y) ((X) + (Y))
3
4  __kernel void marrow_kernel(__global T *g_idata,
5    __global T *g_odata,
6    const unsigned long n,
7    __local volatile T* sdata)
8  {
9      size_t blockSize = get_local_size(0);
10     size_t tid = get_local_id(0);
```

```
11      size_t first = get_group_id(0) * 2 * blockSize + tid;
12      size_t gridSize = blockSize * 2 * get_num_groups(0);
13
14      size_t i = first;
15      if (i < n) {
16         sdata[tid] = g_idata[i];
17
18         if (i + blockSize < n)
19           sdata[tid] = Operation(sdata[tid], g_idata[i + blockSize]);
20         i += gridSize;
21
22         while (i < n) {
23          sdata[tid] = Operation(sdata[tid], g_idata[i]);
24
25           if (i + blockSize < n)
26              sdata[tid] = Operation(sdata[tid], g_idata[i + blockSize]);
27           i += gridSize;
28         }
29      }
30
31      barrier(CLK_LOCAL_MEM_FENCE);
32
33      if (tid < 128 && (first + 128 < n)) {
34        sdata[tid] = Operation(sdata[tid], sdata[tid + 128]);
35      }
36      barrier(CLK_LOCAL_MEM_FENCE);
37
38      if (tid < 64 && (first + 64 < n)) {
39        sdata[tid] = Operation(sdata[tid], sdata[tid + 64]);
40      }
41      barrier(CLK_LOCAL_MEM_FENCE);
42
43      if (tid < 32) {
44         if (first + 32 < n) sdata[tid] =
45          Operation(sdata[tid], sdata[tid + 32]);
46         if (first + 16 < n) sdata[tid] =
47          Operation(sdata[tid], sdata[tid + 16]);
48         if (first +  8 < n) sdata[tid] =
49          Operation(sdata[tid], sdata[tid + 8]);
50         if (first +  4 < n) sdata[tid] =
51          Operation(sdata[tid], sdata[tid + 4]);
52         if (first +  2 < n) sdata[tid] =
53          Operation(sdata[tid], sdata[tid + 2]);
54         if (first +  1 < n) sdata[tid] =
55          Operation(sdata[tid], sdata[tid + 1]);
56      }
```

```
57
58      if (tid == 0)
59          g_odata[get_group_id(0)] = sdata[0];
60  }
```

Listagem A.10: 1º *Kernel* OpenCL para o cálculo do erro de quantização.

```
1  __kernel void marrow_kernel(global float* out_array27,
2    global float* array28,
3    global float* array29,
4    const int int_30,
5    const unsigned long limit0)
6  {
7  const size_t index0 = get_global_id(0);
8  if (index0 >= limit0) return;
9  float local10 = array28[index0] - array29[index0];
10 out_array27[index0] = pow(local10, int_30);
11 }
```

ANEXO I

Anexo 1 Testes unitários

I.1 Testes UbiSOM

Listagem I.1: Teste da definição e obtenção do estado do mapa.

```
1  TEST(UbiSOM, UbiSOM_SetMap_Test) {
2
3      UbiSOM<1, 3, 3, 10> ubisom;
4
5      marrow::matrix<float, 9, 1> map;
6      map.fill(0);
7
8      ubisom.set_map(map);
9
10     const marrow::matrix<float, 9, 1>& rec_map = ubisom.get_map();
11
12     for(int i = 0; i < 9 ; i++)
13         EXPECT_FLOAT_EQ(map[i][0], rec_map[i][0]);
14
15 }
```

Listagem I.2: Teste da obtenção da BMU para mapas com 1 dimensão.

```
1  TEST(UbiSOM, UbiSOM_Process_BMU_Test) {
2
3      //setup ubisom initial state...
4      UbiSOM<1, 3, 3, 10> ubisom;
5
6      marrow::matrix<float, 9, 1> init_map;
7      init_map.fill(0);
```

```
8       init_map[4][0] = 1.0f;
9
10      ubisom.set_map(init_map);
11
12      //setup observation
13      marrow::array<float, 1> obs;
14      obs[0] = 1.0f;
15
16      //test
17      auto bmu = ubisom.process_bmu(obs);
18
19      EXPECT_EQ(1, std::get<0>(bmu));
20      EXPECT_EQ(1, std::get<1>(bmu));
21
22  }
```

Listagem I.3: Teste da obtenção da BMU para mapas com 2 dimensões.

```
1   TEST(UbiSOM, UbiSOM_Process_BMU_2F_Test) {
2
3       //setup ubisom initial state...
4       UbiSOM<2, 3, 3, 10> ubisom;
5
6       marrow::matrix<float, 9, 2> init_map;
7       init_map.fill(0);
8       init_map[4][0] = 1.0;
9       init_map[4][1] = 1.0;
10      init_map[3][0] = 1.0;
11      init_map[3][1] = 0;
12      init_map[5][0] = 0;
13      init_map[5][1] = 1.0;
14
15      ubisom.set_map(init_map);
16
17      //setup observation
18      marrow::array<float, 2> obs;
19      obs.fill(1);
20
21      //test
22      auto bmu = ubisom.process_bmu(obs);
23
24      EXPECT_EQ(1, std::get<0>(bmu));
25      EXPECT_EQ(1, std::get<1>(bmu));
26
27  }
```

Listagem I.4: Teste do 1º Kernel com 1 dimensão.

```
TEST(UbiSOM, UbiSOM_Kernel_1_Test) {

    //setup ubisom initial state...
    UbiSOM<1, 3, 3, 10> ubisom;

    marrow::matrix<float, 9, 1> init_map;
    for(int i = 0; i < 9; i++)
        init_map[i][0] = (i * 1.0f)/9.0f;

    ubisom.set_map(init_map);

    //define observation
    marrow::array<float, 1> obs;
    obs[0] = 0.0f;

    marrow::array<float, 9> distances;
    ubisom.map_distances(obs, distances);

    marrow::array<float, 9> expected;
    expected.fill(
            [](int i) -> float {
                return (float)pow((i * 1.0f)/9.0f, 2);
            });

    for(int i = 0; i < 9; i++)
        EXPECT_FLOAT_EQ(expected[i], distances[i]);


}
```

Listagem I.5: Teste do 1º Kernel com 2 dimensões.

```
TEST(UbiSOM, UbiSOM_Kernel_1_2F_Test) {

    //setup ubisom initial state...
    UbiSOM<2, 3, 3, 10> ubisom;

    marrow::matrix<float, 9, 2> init_map;
    init_map.fill([](int i, int j) { return j == 0 ? (float)i : 0; });

    ubisom.set_map(init_map);

    //define observation
    marrow::array<float, 2> obs;
    obs.fill(0);
```

```
14
15      //define results
16      marrow::array<float, 9> expected_results;
17      expected_results.fill(
18              [&init_map](int i) {
19                  return init_map[i][0] * init_map[i][0];
20              });
21
22
23      marrow::array<float, 9> distances;
24      ubisom.map_distances(obs, distances);
25
26      for(int i = 0; i < 9; i++)
27          EXPECT_EQ(expected_results[i], distances[i]);
28
29
30  }
```

Listagem I.6: Teste do 2º Kernel

```
1   TEST(UbiSOM, UbiSOM_Kernel_2_Test) {
2
3       //setup ubisom initial state...
4       UbiSOM<1, 3, 3, 10> ubisom;
5
6       marrow::matrix<float, 9, 1> init_map;
7       init_map.fill([](int i, int j) -> float {
8           return j == 0 ? (float)i : 0;
9       });
10
11
12      ubisom.set_map(init_map);
13
14      //define distances
15      marrow::array<float, 9> distances;
16      distances[0] = 15.0/3000.0;
17      distances[1] = 9.0/3000.0;
18      distances[2] = 2550.0/3000.0;
19      distances[3] = 99.0/3000.0;
20      distances[4] = 45.0/3000.0;
21      distances[5] = 5.0/3000.0;
22      distances[6] = 3.0/3000.0; //este é o menor...
23      distances[7] = 1000.0/3000.0;
24      distances[8] = 500.0/3000.0;
25
26
27      int bmu = ubisom.get_bmu(distances);
```

```
28      EXPECT_EQ(6, bmu);
29
30  }
```

Listagem I.7: Teste do 3º Kernel, mapa 3x3

```
1   TEST(UbiSOM, UbiSOM_Kernel_3_Test) {
2
3       //setup ubisom initial state...
4       UbiSOM<1, 3, 3, 10> ubisom;
5
6       marrow::matrix<float, 9, 1> init_map;
7       init_map.fill(0);
8       init_map[4][0] = 1.0f; //bmu
9
10      ubisom.set_map(init_map);
11
12      //define observation
13      marrow::array<float, 1> obs;
14      obs[0] = 1.0f;
15
16      ubisom.update_map(4, 1.0f, 1.0f, obs);
17
18      array<float, 9> expected_results;
19      expected_results[0] = expected_results[2] = expected_results[6] =
20       expected_results[8] = 0.89483929;
21      expected_results[1] = expected_results[3] = expected_results[5] =
22       expected_results[7] = 0.94595945;
23      expected_results[4] = 1.0;
24
25
26      const marrow::matrix<float, 9, 1>& res_map = ubisom.get_map();
27
28      for(int i = 0; i < 9; i++) {
29          EXPECT_FLOAT_EQ(expected_results[i], res_map[i][0]);
30      }
31
32  }
```

Listagem I.8: Teste do 3º Kernel, mapa 5x5

```
1   TEST(UbiSOM, UbiSOM_Kernel_3_Test_2) {
2
3       //setup ubisom initial state...
4       UbiSOM<1, 5, 5, 10> ubisom;
5
6       marrow::matrix<float, 25, 1> init_map;
```

```
7       init_map.fill(0);
8       init_map[6][0] = 500.0/1000.0;
9       init_map[7][0] = 500.0/1000.0;
10      init_map[8][0] = 500/1000.0;
11      init_map[11][0] = 500/1000.0;
12      init_map[12][0] = 1000/1000.0; //BMU
13      init_map[13][0] = 500/1000.0;
14      init_map[16][0] = 500/1000.0;
15      init_map[17][0] = 500/1000.0;
16      init_map[18][0] = 500/1000.0;
17
18      ubisom.set_map(init_map);
19
20      //define observation
21      marrow::array<float, 1> obs;
22      obs[0] = 1000/1000.0;
23
24      ubisom.update_map(12, 1.0f, 1.0f, obs);
25
26      //define expected result
27      marrow::array<float, 25> expected_result;
28      expected_result[0] = expected_result[4] = expected_result[20] =
29       expected_result[24] = 0.852144;
30      expected_result[1] = expected_result[3] = expected_result[5] =
31       expected_result[9] = expected_result[15] = expected_result[19]
32          = expected_result[21] = expected_result[23] = 0.9048374;
33      expected_result[2] = expected_result[10] = expected_result[14] =
34       expected_result[22] = 0.923116346;
35      expected_result[6] = expected_result[8] = expected_result[16] =
36       expected_result[18] = 0.98039472;
37      expected_result[7] = expected_result[11] = expected_result[13] =
38       expected_result[17] = 0.99009931;
39      expected_result[12] = 1.0;
40
41      const marrow::matrix<float, 25, 1>& res_map = ubisom.get_map();
42
43      for(int i = 0; i < 25; i++) {
44          EXPECT_FLOAT_EQ(expected_result[i], res_map[i][0]);
45      }
46
47  }
```

Listagem I.9: Teste ao cálculo da utilidade de neurónios

```
1   TEST(UbiSOM, UbiSOM_Neuron_Utility_Test) {
2
3       //setup map...
```

```
4       UbiSOM<1, 3, 3, 2000> ubisom;
5
6       marrow::matrix<float, 9, 1> init_map;
7       init_map.fill(0);
8       init_map[4][0] = 1.0; //bmu
9
10      ubisom.set_map(init_map);
11
12      //setup update array...
13      marrow::array<int, 9> last_update;
14      last_update.fill(
15              [](int i) {
16               return i * 300;
17              }); //0, 300, 600, 900, 1200, 1500, 1800, 2100, 2400
18
19      int t = 3000;
20
21      //get neuron utility
22      float neuron_utility =
23        ubisom.process_neuron_utility(t, last_update);
24
25      float expected_value = 5.0f/9.0f;
26      EXPECT_FLOAT_EQ(expected_value, neuron_utility);
27
28  }
```

Listagem I.10: Teste ao cálculo da quantização de neurónios

```
1   TEST(UbiSOM, UbiSOM_Quantization_Error_Test) {
2
3       //setup map...
4       UbiSOM<1, 3, 3, 2000> ubisom;
5
6       marrow::matrix<float, 9, 1> init_map;
7       init_map.fill(0);
8       init_map[4][0] = 1; //bmu
9
10      ubisom.set_map(init_map);
11
12      //setup observation
13      marrow::array<float, 1> observation;
14      observation[0] = 1;
15
16      float quantization_error =
17        ubisom.process_quantization_error(observation, 4, 1.0);
18
19      EXPECT_EQ(0, quantization_error);
```

```
20 }
```

Listagem I.11: Teste ao cálculo do erro topológico

```
1  TEST(UbiSOM, UbiSOM_Topological_Error_Test)
2  {
3
4      //setup map...
5      UbiSOM<1, 5, 5, 2000> ubisom;
6
7      marrow::matrix<float, 25, 1> init_map;
8      init_map.fill(0);
9      init_map[4][0] = 10.0f/1000.0f; //bmu
10     init_map[3][0] = 5.0f/1000.0f; //2a BMU
11
12     ubisom.set_map(init_map);
13
14     //setup observation
15     marrow::array<float, 1> observation;
16     observation[0] = 10.0f/1000.0f;
17
18     marrow::array<float, 25> distances;
19     distances.fill(0);
20     ubisom.map_distances(observation, distances);
21
22     float topological_error =
23       ubisom.compute_topological_error(4, distances);
24
25     EXPECT_FLOAT_EQ(0.0005f, topological_error);
26 }
```

Listagem I.12: Teste ao método de treino.

```
1  TEST(UbiSOM, UbiSOM_Train_Test_1)
2  {
3
4      UbiSOM<1, 3, 3, 2000> ubisom;
5      marrow::matrix<float, 9, 1> init_map;
6
7      init_map.fill(
8              [](int i, int j) -> float {
9                  return (i + j) * 1000.0f;
10             });
11
12     ubisom.set_map(init_map);
13     ubisom.set_parameters(1.0f, 0.05f, 1.0f, 0.05f, 0.5f);
14
```

```
15      std::list<marrow::array<float, 1>> observations;
16      for(int i = 0; i < 3; i++)
17      {
18          observations.emplace_back();
19          marrow::array<float, 1>& obs = observations.back();
20          obs[0] = (i + 1) * 1000.0f;
21      }
22
23        ubisom.train(observations);
24
25      marrow::matrix<float, 9, 1> expected_result;
26      expected_result[0][0] = 2544.2253;
27      expected_result[1][0] = 2704.4807;
28      expected_result[2][0] = 2750.4258;
29      expected_result[3][0] = 2733.186;
30      expected_result[4][0] = 2886.5127;
31      expected_result[5][0] = 2939.2751;
32      expected_result[6][0] = 2799.6729;
33      expected_result[7][0] = 2942.8301;
34      expected_result[8][0] = 2997.1235;
35
36      const marrow::matrix<float, 9, 1>& result = ubisom.get_map();
37
38      for(int i = 0; i < 9; i++)
39          EXPECT_FLOAT_EQ(expected_result[i][0], result[i][0]);
40  }
```

I.2 Testes Marrow

Listagem I.13: Teste da implementação do *Kernel* OpenCL de redução de matriz para array.

```
1
2   TEST(Marrow, MatrixToArrayReduction) {
3
4       int uploads = profiler::number_uploads();
5       int downloads = profiler::number_downloads();
6
7       matrix<float, 800, 4> map;
8       map.fill([](int i, int j) { return (float)i; });
9
10      array<float, 800> result;
11      array<int, 800 * 4> debug;
12
13      long average = 0;
```

```
14      for(int i = 0; i < 50; i++) {
15
16          auto start = std::chrono::steady_clock::now();
17
18          kernel<float &, float, int &, unsigned long, unsigned long,
19          local<int, 128>>(
20                  "reduction_matrix_float_plus.cl", "marrow_kernel",
21                  {128})(result, map, debug, 4, 800, 0);
22
23          auto end = std::chrono::steady_clock::now();
24          auto time = end - start;
25          average = (average +
26              std::chrono::duration_cast<std::chrono::nanoseconds>(time)
27                  .count()) / 2;
28
29      }
30
31      REPORT("Executed␣in␣" << average << "␣ns.");
32
33      marrow::array<float, 800> expected_result;
34      expected_result.fill([](int i){return (float)(4 * i);});
35
36      for(int i = 0; i < 800; i++) {
37          EXPECT_EQ(expected_result[i], result[i]);
38      }
39
40      REPORT("Uploads:␣" << profiler::number_uploads() - uploads
41          << "␣Downloads:␣"
42          << profiler::number_downloads() - downloads);
43
44  }
```

Listagem I.14: Teste da implementação do *Kernel* Marrow de redução de matriz para array.

```
1
2   TEST(Marrow, MarrowMatrixToArrayReduction) {
3
4       int uploads = profiler::number_uploads();
5       int downloads = profiler::number_downloads();
6
7       matrix<float, 800, 4> map;
8       map.fill([](int i, int j) { return (float)i; });
9
10      array<float, 800> result;
11      long average = 0;
```

```
12      for(int i = 0; i < 50; i++) {
13
14          auto start = std::chrono::steady_clock::now();
15
16          result = reduce<plus<float>>()(map);
17
18          auto end = std::chrono::steady_clock::now();
19          auto time = end - start;
20          average = (average +
21              std::chrono::duration_cast<std::chrono::nanoseconds>(time)
22                  .count()) / 2;
23
24      }
25
26      REPORT("Executed in " << average << " ns.");
27
28      marrow::array<float, 800> expected_result;
29      expected_result.fill([](int i){return (float)(4 * i);});
30
31      for(int i = 0; i < 800; i++) {
32          EXPECT_EQ(expected_result[i], result[i]);
33      }
34
35      REPORT("Uploads: " << profiler::number_uploads() - uploads
36          << " Downloads: "
37          << profiler::number_downloads() - downloads);
38
39  }
```

Listagem I.15: Teste da implementação do *Kernel* Marrow de redução de matriz para array para número de colunas par não potências de 2.

```
1   TEST(Marrow, Non2MultiMatrixToArrayReduction) {
2
3       int uploads = profiler::copy_out_ops();
4       int downloads = profiler::copy_in_ops();
5
6       const size_type rows = 800;
7       const size_type columns = 10;
8
9       matrix<float, rows, columns> map;
10      map.fill([](int i, int j) { return (float)i; });
11
12      array<float, rows> result;
13
14      kernel<float &, float, unsigned long, unsigned long,
```

```
        local<float, 128>>(
            "reduction_2_to_1_float_plus.cl", "marrow_kernel",
            {128}, { rows * columns })
        (result, map, columns, rows, 0);

    marrow::array<float, rows> expected_result;
    expected_result.fill([](int i){return (float)(columns * i);});

    for(int i = 0; i < rows; i++) {
        EXPECT_FLOAT_EQ(expected_result[i], result[i]);
    }


    REPORT("Uploads: " << profiler::copy_out_ops() - uploads <<
    " Downloads: " << profiler::copy_in_ops() - downloads);

}
```

Listagem I.16: Teste da implementação do *Kernel* Marrow de redução de matriz para array para número de colunas impar.

```
    TEST(Marrow, ImparMatrixToArrayReduction) {

        int uploads = profiler::copy_out_ops();
        int downloads = profiler::copy_in_ops();

        const size_type rows = 800;
        const size_type columns = 5;

        matrix<float, rows, columns> map;
        map.fill([](int i, int j) { return (float)i; });

        array<float, rows> result;

        kernel<float &, float, unsigned long, unsigned long,
        local<float, 128>>(
                "reduction_2_to_1_float_plus.cl", "marrow_kernel",
                {128}, { rows * columns })
            (result, map, columns, rows, 0);

        marrow::array<float, rows> expected_result;
        expected_result.fill([](int i){return (float)(columns * i);});

        for(int i = 0; i < rows; i++) {
            EXPECT_FLOAT_EQ(expected_result[i], result[i]);
        }
```

```
26
27          REPORT("Uploads: " << profiler::copy_out_ops() - uploads
28          << " Downloads: " << profiler::copy_in_ops() - downloads);
29
30      }
```

Listagem I.17: Teste da implementação do *Kernel* Marrow de redução de matriz para array com 1000 colunas.

```
1       TEST(Marrow, BigMatrixToArrayReduction) {
2
3           int uploads = profiler::copy_out_ops();
4           int downloads = profiler::copy_in_ops();
5
6           const size_type rows = 800;
7           const size_type columns = 1000;
8
9           matrix<float, rows, columns> map;
10          map.fill([](int i, int j) { return (float)i; });
11
12          array<float, rows> result;
13
14          kernel<float &, float, unsigned long, unsigned long,
15              local<float, 128>>(
16                  "reduction_2_to_1_float_plus.cl", "marrow_kernel",
17                  {128}, { rows * columns })
18              (result, map, columns, rows, 0);
19
20          marrow::array<float, rows> expected_result;
21          expected_result.fill([](int i){return (float)(columns * i);});
22
23          for(int i = 0; i < rows; i++) {
24              EXPECT_FLOAT_EQ(expected_result[i], result[i]);
25          }
26
27
28          REPORT("Uploads: " << profiler::copy_out_ops() - uploads
29          << " Downloads: " << profiler::copy_in_ops() - downloads);
30
31      }
```

ANEXO II

Anexo 2 *Scripts* Auxiliares

Listagem II.1: *Script* em *Python* para normalização de um *dataset*.

```
import numpy as np
import sys
from sklearn.preprocessing import MinMaxScaler

#setup stuff...

read_data = np.genfromtxt(input, delimiter=',', names=True)
l = 0
if ignore_first_row:
    l = 1
c = 0
if ignore_first_column:
    c = 1

data = np.zeros((len(read_data) - l, len(read_data[0]) - c))
for i in range(l, len(read_data)):
    tuple = read_data[i]
    for j in range(c, len(tuple)):
        data[i - l][j - c] = tuple[j]

scaler = MinMaxScaler(copy=True, feature_range=(0, 1)).fit(data)
scalled_data = scaler.transform(data)
np.savetxt(output, scalled_data, delimiter=",")
```

Listagem II.2: *Script* em *Python* para expansão de um *dataset*.

```
import sys
from numpy.random import uniform

#setup stuff...

lines = []
i = 0
with open(input_file, 'r') as in_file:
    for line in in_file:
        if ignore_first_row and i == 0:
            i = i + 1
            continue
        else:
            i = i + 1
            lines.append(line)

with open(output_file, 'w') as out_file:
    i = 0
    while i < new_size:
        pick = int(uniform() * len(lines))
        out_file.write(lines[pick])
        i = i + 1
```

Listagem II.3: *Script* em *Python* para gerar mapa inicial

```

import sys
import numpy as np

lines = int(sys.argv[1])
cols = int(sys.argv[2])
features = int(sys.argv[3])
output = sys.argv[4]

data = np.random.random_sample((lines * cols, features))

np.savetxt(output, data, delimiter=",")
```

ANEXO III

ANEXO 3 *Kernels* OPENCL

Listagem III.1: *Kernel* OpenCL para redução de matriz a *array* (Parte 1).

```
1  inline void atomicAdd_g_f(volatile __global T *addr, T val)
2  {
3      union {
4          unsigned int u32;
5          float        f32;
6      } next, expected, current;
7  current.f32    = *addr;
8      do {
9      expected.f32 = current.f32;
10         next.f32     = expected.f32 + val;
11     current.u32  = atomic_cmpxchg(
12         (volatile __global unsigned int *)addr,
13         expected.u32, next.u32);
14     } while( current.u32 != expected.u32 );
15 }
16
17 __kernel void marrow_kernel( __global T *g_odata,
18     __global T *g_idata, const unsigned long column_count,
19     const unsigned long row_count, __local volatile T* sdata)
20 {
21     size_t tid = get_local_id(0);
22     size_t gid = get_global_id(0);
23     size_t row = gid / column_count;
24     size_t column = gid % column_count;
25     bool group_start = false;
```

Listagem III.2: *Kernel* OpenCL para redução de matriz a *array* (Parte 2).

```
    if(row < row_count && column < column_count)
    {
        sdata[tid] = g_idata[gid];
    }
    barrier(CLK_LOCAL_MEM_FENCE);

    if(row < row_count && column < column_count)
    {
        //see if the sum is split between work-groups or not....
        size_t first_element_pos = (row * column_count)
            % get_local_size(0);
        size_t last_element_pos = first_element_pos + column_count - 1;
        //group is divided between workgroups
        if(last_element_pos >= get_local_size(0)) {
            size_t head_remain_units = get_local_size(0) - first_element_pos;
            if(column < head_remain_units) { //first work-group
                size_t step = head_remain_units / 2;
                size_t limit = head_remain_units;
                while(step > 0)
                {
                    if(column + step < limit)
                        sdata[tid] += sdata[tid + step];
                    barrier(CLK_LOCAL_MEM_FENCE);
                    if(column == 0 && limit % 2 == 1)
                        sdata[tid] += sdata[tid + limit - 1];
                    barrier(CLK_LOCAL_MEM_FENCE);
                    step /= 2;
                    limit /= 2;
                }
                group_start = column == 0;
            } else {
                size_t other_units_count = column_count
                    - head_remain_units;
                size_t other_work_groups_count = other_units_count
                    / get_local_size(0);
                size_t mid_units_count = other_work_groups_count
                    * get_local_size(0);
                size_t tail_start = head_remain_units
                    + mid_units_count;
                size_t tail_size = column_count - tail_start;
                if(column >= tail_start) { // last work group
                    size_t limit = tail_size;
                    size_t step = tail_size / 2;
                    while(step > 0) {
                        if(tid + step < limit)
                            sdata[tid] += sdata[tid + step]
                        barrier(CLK_LOCAL_MEM_FENCE);
                        if(tid == 0 && limit % 2 == 1)
                            sdata[tid] += sdata[tid + limit - 1];
                        barrier(CLK_LOCAL_MEM_FENCE);
                        step /= 2;
                        limit /= 2;
                    }
                    group_start = tid == 0;
```

Listagem III.3: *Kernel* OpenCL para redução de matriz a *array* (Parte 3).

```
            } else {// remaining work groups, lets assume it's always
                    //a multiple of 2
            size_t step = get_local_size(0) / 2;
            size_t limit = get_local_size(0);
            while(step > 0)
            {
                if(tid + step < limit)
                    sdata[tid] += sdata[tid + step];
                barrier(CLK_LOCAL_MEM_FENCE);
                step /= 2;
                limit /= 2;
            }
            group_start = tid == 0;
            }
        }
    } else { //caso comum
    size_t step = column_count / 2;
    size_t limit = column_count;
    while(step > 0) {
        if(column + step < limit)
            sdata[tid] += sdata[tid + step];
        barrier(CLK_LOCAL_MEM_FENCE);
        if(column == 0 && limit % 2 == 1)
            sdata[tid] += sdata[tid + limit - 1];
        barrier(CLK_LOCAL_MEM_FENCE);
        step /= 2;
        limit /= 2;
    }
    group_start = column == 0;
    }
}
barrier(CLK_LOCAL_MEM_FENCE);
if(row < row_count && group_start)
{
    atomicAdd_g_f((g_odata + row), sdata[tid]);
}
}
```

ANEXO IV

Anexo 4 Mensagens Protobuf

Listagem IV.1: Definição em *protobuf* da mensagem *Init*

```
message Init {
  optional float initial_learning_rate = 1;
  optional float final_learning_rate = 2;
  optional float initial_normalized_radius = 3;
  optional float final_normalized_radius = 4;
  optional float beta = 5;
  optional float omega = 6;

  repeated double initial_map = 7;
}
```

Listagem IV.2: Definição em *protobuf* da mensagem *InitAck*

```
message InitAck
{
    required bool success = 1;
}
```

Listagem IV.3: Definição em *protobuf* da mensagem *Train*

```
message Train {
  repeated float observations = 1;
}
```

Listagem IV.4: Definição em *protobuf* da mensagem *Map*

```
message Map {
  required int32 t = 1;
```

```
3
4     optional float average_neuron_utility = 2;
5     optional float average_quantization_error = 3;
6     optional float drift = 4;
7
8     repeated float map = 5;
9 }
```

Listagem IV.5: Definição em *protobuf* da mensagem *SetMap*

```
1
2 message SetMap {
3     repeated float new_map = 1;
4 }
```

Listagem IV.6: Definição em *protobuf* da mensagem *SetMapAck*

```
1 message SetMapAck
2 {
3     required bool success = 1;
4 }
```

Listagem IV.7: Definição em *protobuf* das mensagens *GetLastBmu* e *LastBmu*

```
1 message GetLastBmu {}
2
3 message LastBmu
4 {
5     required int32 x = 1;
6     required int32 y = 2;
7 }
```

Listagem IV.8: Definição em *protobuf* das mensagens *GetLastBmu* e *LastBmu*

```
1 message RequestBmuProcess
2 {
3     repeated float observation = 1;
4 }
5
6 message ProcessedBmu
7 {
8     required int32 x = 1;
9     required int32 y = 2;
10 }
```

Listagem IV.9: Definição em *protobuf* da mensagem de Erro.

```
1 message Error
```

```
2 {
3     optional int32 error_type = 1;
4     optional string error_message = 2;
5 }
```

ANEXO V

Anexo 4 Alterações ao Documento

Melhorias do estilo de escrita e correções ortográficas ao longo da dissertação.

Adicionou-se linha nova na tabela 3.2 contendo informações sobre a coleção que contém o mapa UbiSOM. Alterou-se descrições do tipo de dados usado em cada coleção.

Clareou-se significado das numerações (e letras) usadas na figura 3.10.

Adicionou-se segmento na secção 3.2.4.1 para justificar como se poderia melhorar o cálculo do erro topológico.

Dividiu-se secção de Metodologia em duas secções distintas: Métricas de Avaliação (4.1) e Metodologia (4.2).

Completou-se tabela 4.1 com número de *threads* de *hardware* dos processadores em questão.

Adicionou-se sub-secção 4.3.2.6, com resumo dos resultados obtidos em relação á análise visual do resultado do processamento de cada *dataset*.

Corrigiu-se apresentação dos gráficos 4.12, 4.13 e 4.14

Alterou-se o titulo do capítulo 5. Alterou-se vários elementos do texto relacionado com o capítulo (incluindo o texto da secção 5.1, Trabalho Futuro).

Adicionou-se referências [15, 61, 70].

Corrigiu-se formatação da listagem A.7.