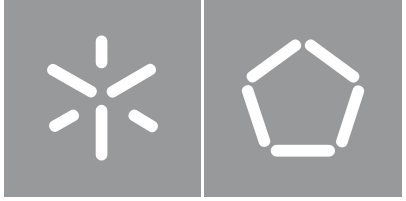
Universidade do Minho
Escola de Engenharia

Diana Abreu Machado Azevedo Rodrigues

Controlo energético de GPU em sistemas de Deep Learning

janeiro 2025



Universidade do Minho
Escola de Engenharia

Diana Abreu Machado Azevedo Rodrigues

Controlo energético de GPU em sistemas de Deep Learning

Dissertação de Mestrado
Mestrado em Engenharia Informática

Trabalho efetuado sob a orientação de

Ricardo Gonçalves Macedo (Orientador)
António Luís Pinto Ferreira Sousa (Co-orientador)
Cláudia Vanessa Martins de Brito (Supervisora no Centro de Investigação)

janeiro 2025

Direitos de Autor e Condições de Utilização do Trabalho por Terceiros

Este é um trabalho académico que pode ser utilizado por terceiros desde que respeitadas as regras e boas práticas internacionalmente aceites, no que concerne aos direitos de autor e direitos conexos.

Assim, o presente trabalho pode ser utilizado nos termos previstos na licença abaixo indicada.

Caso o utilizador necessite de permissão para poder fazer um uso do trabalho em condições não previstas no licenciamento indicado, deverá contactar o autor, através do RepositóriUM da Universidade do Minho.

Licença concedida aos utilizadores deste trabalho:

CC BY 4.0

https://creativecommons.org/licenses/by/4.0/

Agradecimentos

Com esta dissertação, encerro a minha jornada na Universidade do Minho, foram anos desafiantes mas repletos de aprendizagens. O apoio incondicional das pessoas que me acompanharam ao longo deste percurso foi indispensável e, por isso, a cada uma delas, expresso a minha gratidão.

Primeiramente, gostaria de agradecer aos meus orientadores, Professor Ricardo Macedo, Professora Cláudia Brito e Professor António Luís Sousa. Foram fundamentais neste percurso e não poderia ter escolhido melhores orientadores para me acompanharem. Agradeço-vos a paciência, disponibilidade e orientação constante.

Quero agradecer também à minha família que me apoiou incondicionalmente e garantiu que sempre tive as melhores condições para alcançar todos os meus objetivos.

Deixo ainda uma palavra de agradecimento aos meus amigos de infância. Obrigada por acreditarem em mim e me motivarem mesmo sem compreenderem exatamente em que consiste a minha dissertação.

Por fim, mas não menos importante agradeço, à Mariana, à Sara e ao Zé com quem partilhei esta jornada académica. A vossa companhia tornou estes anos mais fáceis, mas acima de tudo divertidos e, sem dúvida, memoráveis.

Este trabalho foi financiado por Fundos Nacionais através da agência de financiamento portuguesa, FCT - Fundação para a Ciência e a Tecnologia, no âmbito do projeto LA/P/0063/2020. DOI 10.54499/LA/P/0063/2020. e cofinanciado pela Componente 5 - Capitalização e Inovação Empresarial, integrada na Dimensão Resiliência do Plano de Recuperação e Resiliência no âmbito do Mecanismo de Recuperação e Resiliência (MRR) da União Europeia (UE), enquadrado no Próximo Geração EU, para o período 2021 - 2026, no âmbito do projeto ATE, com referência 56. A FCT e a FCCN financiaram ainda esta tese via da utilização da plataforma Deucalion através do projeto com referência única 2024.00014.TEST.DEUCALION.

Declaração de Integridade

Declaro ter atuado com integridade na elaboração do presente trabalho académico e confirmo que não recorri à prática de plágio nem a qualquer forma de utilização indevida ou falsificação de informações ou resultados em nenhuma das etapas conducente à sua elaboração.

Mais declaro que conheço e que respeitei o Código de Conduta Ética da Universidade do Minho.

Universidade do Minho, Braga, janeiro 2025

Diana Abreu Machado Azevedo Rodrigues

Resumo

A área da **Aprendizagem Profunda** sofreu uma enorme expansão nos últimos anos, com modelos cada vez mais complexos e *datasets* cada vez maiores. Devido à crescente complexidade dos modelos, foi adotado o uso das **GPU**s para o seu treino, tornando este processo mais eficiente. No entanto, o crescimento progressivo do poder de processamento destes dispositivos causou um aumento do seu consumo energético.

Para além da **GPU**, o processo de treino de um modelo de **AP** utiliza todos os outros recursos existentes num servidor: disco, **RAM** e **CPU**, podendo ainda recorrer à rede no caso de treino distribuído e nos casos em que o *dataset* está em armazenamento remoto. Quando o gargalo é qualquer outro recurso que não a **GPU** significa que o treino dos fragmentos neste recurso é mais rápido do que a reposição de novos fragmentos, ficando a **GPU** inutilizada até ao momento em que esse carregamento é concluído. Por padrão, a **GPU** opera à frequência máxima de forma a terminar o treino o mais rápido possível mas, nesta situação, acaba por ficar inativa. Por atuar a alta frequência apresenta um elevado consumo energético durante o treino, consumindo também energia durante o período de inutilização. Através do ajuste de frequência da **GPU** é possível diminuir o custo energético associado ao treino de modelos. No entanto, este ajuste deverá ter em conta as características da carga de trabalho por forma a não degradar o seu tempo de execução.

De forma a resolver estes problemas, esta dissertação apresenta o WAFT, um novo sistema de otimização energética de **GPU**s especificamente concebido para treino de modelos de **AP**. Este sistema recorre ao ajuste dinâmico da frequência da **GPU** reconhecendo as várias fases de uma iteração. Adicionalmente, adapta o seu comportamento às característica do modelo e do *hardware*, de forma a diminuir o consumo energético sem prejudicar o tempo de execução.

Através de uma avaliação experimental compreensiva, mostramos que o WAFT consegue diminuir o consumo energético até 42% em relação aos cenários usados em ambientes de produção. Ainda, quando comparado com soluções do estado da arte, o WAFT apresenta ganhos de poupança energética até 29%.

Palavras-chave Otimização do Consumo Energético, Aprendizagem Profunda

Abstract

In recent years, **Deep Learning (DL)** has experienced massive growth, using increasingly complex models and larger datasets. Due to the rising complexity of models, the use of **GPU**s to train these has become prevalent due to their efficiency in the training process. However, the progressive growth of the processing capabilities of these devices came at the cost of higher energy consumption.

Besides using the **GPU**, the training process of a **DL** model also uses the remaining resources in a server: disk, **RAM** and **CPU**, it may even resort to the network when it comes to distributed training or when the dataset is stored remotely. When the bottleneck is a resource other than the **GPU**, the training of samples on this resource is faster than the loading of new samples, leaving the **GPU** idle until the moment the loading is complete. By default, the **GPU** operates at its maximum frequency in order to finish the training as fast as possible, however, in this situation, it comes to a halt. As a result of running at maximum frequency, it exhibits significant energy consumption during training, even under idle periods. While adjusting the **GPU** frequency allows decreasing the energy consumption associated with model training, this adjustment must take into consideration the workload's characteristics to guarantee the training time doesn't increase.

To address these problems, this dissertation proposes WAFT, a novel **GPU** energy optimization system specifically designed for **DL** model training. It dynamically adjusts the **GPU** frequency by recognising the different phases of an iteration. Additionally, it adapts its behaviour to the model and hardware characteristics in order to decrease the energy consumption without compromising the execution time.

Through a comprehensive experimental evaluation, we show that WAFT decreases training energy consumption up to 42% when compared to scenarios used in production environments. Furthermore, when compared to state-of-the-art solutions, WAFT improves energy savings up to 29%.

Keywords Energy Consumption Optimization, Deep Learning

Conteúdo

Lista de Figuras

Lista de Tabelas

Acrónimos

AM Aprendizagem Máquina.

AP Aprendizagem Profunda.

CPU Central Processing Unit.

CV Coeficiente de Variação.

DDR Double Data Rate.

DL Deep Learning.

DRS Dynamic Resource Sleep.

DVFS Dynamic Voltage and Frequency Scaling.

GPU Graphics Processing Unit.

NVMe Non-Volatile Memory Express.

RAM Random Access Memory.

RNA Redes Neuronais Artificiais.

RNC Redes Neuronais Convolucionais.

SATA Serial Advanced Technology Attachment.

SM Streaming Multiprocessor.

Capítulo 1

Introdução

Na última década, a utilização de **Aprendizagem Profunda (AP)** aumentou significativamente, sendo hoje crucial para diversas empresas e instituições. Esta permite extrair conhecimento a partir de dados de modo a fornecer um serviço personalizado e de maior qualidade, tornando-o assim mais apelativo para os utilizadores. Um exemplo prático é o desbloqueio de dispositivos e aplicações através de reconhecimento facial, que torna o processo mais ágil e conveniente. Outro caso notável são as cirurgias realizadas por máquinas supervisionadas por médicos, reduzindo significativamente o risco de erro humano numa área onde a precisão é de importância crítica.

AP consiste no treino de um modelo baseado em redes neuronais que requer a utilização de dados pertencentes a um *dataset* (p.ex., fotografias, texto, ficheiros de áudio, etc.). Ao longo dos últimos anos, os conjuntos de dados utilizados para este fim têm-se tornado progressivamente maiores, pois, quanto maiores forem, mais conhecimento é possível extrair através deles dando origem a um modelo potencialmente melhor. Isto é, maiores *datasets* permitem obter melhor acurácia e aumentam a capacidade do modelo de ser generalizável para casos desconhecidos [25, 44].

As soluções baseadas em **AP** seguem uma *pipeline* típica que consiste na coleção e tratamento dos dados, seguida do treino e inferência. Para suportar este processo, os *datasets* utilizados são guardados no sistema de armazenamento, o qual pode ser remoto ou local [20]. Assim, antes de começar o treino do modelo, é necessário que a **Central Processing Unit (CPU)** carregue fragmentos desse *dataset* para a memória para serem pré-processados. Após esse pré-processamento, os fragmentos estão prontos para serem utilizados, um após o outro, no treino do modelo que consiste em aplicar computações (p.ex., multiplicações matriciais, aplicação de funções de ativação e cálculo de gradientes) sucessivamente aos dados do fragmento [26].

Tanto a **CPU** como a **Graphics Processing Unit (GPU)** são capazes de aplicar estas computações, no entanto, tem-se registado um progresso expressivo do poder de processamento das **GPU**s e, por isso, estas têm sido adotadas como padrão para tarefas de **AP**, por garantirem mais eficiência na perspetiva

do desempenho [43].

Para além disso, este processo pode ser executado em apenas uma **GPU** ou então ser distribuído por várias (e até por diferentes nodos computacionais). Essa distribuição pode ser feita recorrendo a diferentes técnicas de paralelismo como paralelismo de dados em que as diferentes **GPU**s aplicam simultaneamente as mesmas computações sobre diferentes conjuntos de dados, ou paralelismo de *pipeline* em que cada **GPU** é responsável por aplicar uma parte das computações. No entanto, independentemente da técnica utilizada, o uso de várias **GPU**s implica comunicação e sincronização entre elas [39].

Assim, o processo de treino de um modelo de **AP** utiliza todos os recursos de um servidor: disco, **Random Access Memory (RAM)**, **CPU** e **GPU** e pode ainda recorrer à rede. Isto significa que o desempenho do treino depende do desempenho, utilização e disponibilidade destes recursos.

Por exemplo, devido à sequencialidade de muitos destes processos, quando as leituras do dispositivo de armazenamento apresentam um gargalo, as fases de pré-processamento e treino serão atrasadas pois, para que sejam iniciadas, precisam dos dados em memória. Isto pode acontecer devido à disparidade do desempenho do dispositivo de armazenamento e a cadência com que a **GPU** processa dados ou, nos casos em que este é remoto, devido a sobrecargas da rede ou contenção causada por acessos concorrentes. Essas fases do treino podem também ser atrasadas quando o espaço em memória não é suficiente, pois os fragmentos contidos na memória são pré-processados e treinados mais rápido do que são repostos novos fragmentos. Em todos estes casos, a **CPU** e a **GPU** ficam à espera de dados para processar, levando a um subaproveitamento destes recursos [25].

Por outro lado, quando o pré-processamento é demorado, a **GPU** acaba por ficar à espera de dados vindos da **CPU** [25]. Este período de espera verifica-se também quando os fragmentos de dados que podem ser acomodados no espaço disponível na memória da **GPU** são treinados mais rapidamente do que conseguem ser trazidos novos fragmentos. Todavia, quando o treino é computacionalmente intensivo (isto é, grande parte do tempo de treino é gasto nas computações associadas à rede neuronal), o gargalo é a **GPU** e, só melhorando o seu desempenho ou aumentando o número de unidades de processamento, é possível diminuir o tempo de execução.

Sendo assim, dependendo do modelo em causa e dos recursos disponíveis, cada um dos recursos supramencionados pode ser considerado um gargalo do treino de um modelo de **AP**.

Desde que a utilização de **GPU**s se tornou uma prática comum no contexto de **AP**, foram feitos vários estudos que tentam melhorar o seu desempenho para os cenários em que este recurso é o gargalo do treino. Recentemente têm-se constatado que, com o enorme poder computacional das **GPU**s, surgiu o problema da eficiência energética [44]. Contudo, nos últimos anos, devido à crescente preocupação

ambiental, têm surgido algumas abordagens no sentido de analisar e melhorar a eficiência energética destes componentes, começando a crescer um tópico de investigação ainda pouco explorado.

Nesses estudos, tem-se tornado evidente que o consumo energético tem aumentado para níveis extraordinários. Por exemplo, o treino do modelo GPT-3 [34] consome o equivalente a 110 anos do consumo energético médio de uma casa em Portugal com base nos dados de 2022.[1]

Estes custos têm-se revelado particularmente notórios no contexto de infraestruturas de grande escala (por exemplo, supercomputadores e centros de dados especializados em Inteligência Artificial) que acomodam milhares de **GPU**s e emitem centenas de toneladas de CO_2 no treino de um único modelo, como o GPT-3 [34]. Consequentemente, também é nesses contextos que as otimizações energéticas têm um maior impacto pois, qualquer percentagem de diminuição do consumo energético, mesmo que pequena, traduz-se numa diminuição considerável das emissões de CO_2.

1.1 Problema

Como referido anteriormente, o desempenho das **GPU**s tem vindo a crescer, no entanto, o desempenho dos restantes recursos não está a acompanhar esse crescimento, sendo esta discrepância cada vez mais evidente [2, 37]. Por estas razões e pelo crescimento constante do tamanho dos *datasets*, o gargalo do treino de um modelo de **AP** muitas vezes não é a **GPU**, o que impede que esta tenha utilização máxima (problemas de subutilização) e, consequentemente, faz com que o tempo de execução seja maior do que poderia ser se contasse com maior desempenho e disponibilidade dos restantes recursos.

Ainda que existam otimizações de armazenamento como *caching/tiering* ou *prefetching* [11, 20, 25, 45], otimizações para o pré-processamento [28, 21] e até soluções para melhor gestão de memória [21], estas não resolvem o problema descrito, já que o tamanho dos *datasets* irá continuar a seguir a tendência atual de crescimento.

Para além do maior tempo de execução, estes problemas levam também a um consumo energético ineficiente, pois a **GPU** consome energia para alcançar o menor tempo de treino possível, no entanto, acaba por ficar inativa à espera de novos dados e, mesmo inutilizada, consome energia [36]. Por esta razão considera-se que, quando o gargalo não é a computação na **GPU**, o treino de modelos é ineficiente em termos energéticos. Isto deve-se, em parte, ao facto de, por padrão, a **GPU** atuar sempre à frequência máxima [13]. Esta configuração permite alcançar o menor tempo de execução de cada computação aplicada pela **GPU**, contudo, leva a um grande consumo energético. Assim, ajustando a frequência,

[1] https://www.ine.pt/xportal/xmain?xpid=INE&xpgid=ine_indicadores&indOcorrCod=0008228&contexto=bd&selTab=tab2

podemos obter diferentes valores de tempo de execução e consumo energético, apesar de não existir uma relação linear entre estas métricas [8, 43]. Ainda que a frequência máxima garanta um menor tempo de execução, mas um consumo energético elevado, não é verdade que a frequência mínima leve a um menor consumo energético. É certo que, com a frequência mínima, é consumida menos energia por unidade de tempo. No entanto, devido à diminuição significativa do desempenho da **GPU**, o tempo de execução aumenta consideravelmente e, consequentemente, o consumo energético total do treino acaba por ser ainda maior do que quando é utilizada a frequência máxima.

Desta forma, quando o gargalo não é a computação, a **GPU** não necessita de operar à frequência máxima, pelo que podemos prolongar o tempo de computação até ao momento em que existem novos dados disponíveis, levando à redução do consumo energético sem prejuízo significativo para o tempo de execução. No entanto, o ajuste da frequência produz diferentes efeitos de acordo com o modelo a treinar, pelo que manipular a frequência da **GPU** de forma agnóstica às características da carga de trabalho pode degradar o tempo de treino.

1.2 Objetivo e Contribuições

Baseado no problema exposto anteriormente, esta dissertação tem como objetivo principal reduzir o consumo energético associado ao treino de modelos de **Aprendizagem Profunda** sem denegrir o desempenho de métricas chave que o caracterizam, como o tempo de treino, de acordo com a carga de trabalho em questão.

De modo a atingir o objetivo supramencionado, esta dissertação propõe três contribuições principais:

- Como primeira contribuição, foi conduzido um estudo experimental de modo a analisar o impacto de diferentes configurações no tempo de treino e no consumo energético da **GPU**. Em detalhe, a análise foi feita sobre diferentes modelos de **AP** (p.ex., AlexNet, ResNet101 e VGG19) utilizando diferentes configurações de frequências de operação. Este estudo mostra que diferentes modelos de **AP**, assim como diferentes fases dentro de uma iteração desse treino, reagem de forma diferente ao ajuste da frequência da **GPU**, o que nos permite concluir que não existe uma única frequência ótima que consiga satisfazer os requisitos de desempenho e consumo energético do treino.

- Como segunda contribuição, com base nos resultados do estudo supramencionado, apresentamos o desenho e implementação do WAFT, um novo sistema de controlo energético de **GPU**s especificamente concebido para treino de modelos de **AP**. Ao contrário do estado da arte, o WAFT

é sensível às características específicas do modelo, ajustando a frequência à qual a **GPU** opera de acordo com a carga de trabalho e, assim, reduzindo o consumo energético sem denegrir o desempenho do treino.

- Como contribuição final, foi realizada uma avaliação compreensiva que demonstra os benefícios do WAFT sobre diferentes ambientes de teste. Em detalhe, o WAFT foi avaliado com diferentes modelos sobre diferentes tipos de **GPU**s e dispositivos de armazenamento, e comparado com soluções do estado da arte. Os resultados mostram que o WAFT consegue reduzir o consumo energético do treino até 42% e 29% quando comparado com o PyTorch nativo e o GEEPAFS [48], sem afetar o tempo de treino.

1.2.1 Resultados

O resultado do trabalho realizado é um repositório público onde está disponibilizado o WAFT [2], sistema que contribui para o avanço do estado da arte relativo ao controlo energético de **GPU**s em sistemas de aprendizagem profunda. Este sistema permite fazer um controlo energético mais granular recorrendo à manipulação da frequência da **GPU**, o que permite identificar e aproveitar períodos de inutilização para melhorar a eficiência energética do treino de modelos de **AP**.

1.3 Estrutura do Documento

Este documento é composto por seis capítulos. O primeiro capítulo corresponde à Introdução, que contextualiza o problema a resolver e aponta o objetivo da dissertação, bem como as suas contribuições. No segundo capítulo, Estado da Arte, é dado o enquadramento necessário ao entendimento do problema e são apresentados alguns trabalhos que se relacionam com esta dissertação.

De seguida, no capítulo Estudo Motivacional, são apresentados resultados do estudo conduzido que sustentaram as decisões tomadas para o desenho do sistema. Desenho esse que é apresentado em detalhe no capítulo seguinte, WAFT. Segue-se a Avaliação, onde é demonstrada a eficiência do sistema no que diz respeito aos objetivos que se pretendia atingir.

Por fim, o último capítulo, Conclusões e trabalho futuro, resume as contribuições e as conclusões retiradas do trabalho desenvolvido, apresentando ainda possíveis caminhos a seguir, de forma a enriquecer a solução proposta.

[2] https://github.com/dsrhaslab/waft

Capítulo 2

Estado da Arte

Neste capítulo são apresentados alguns conceitos da área de **Aprendizagem Profunda (AP)** por forma a enquadrar e contextualizar o problema. Para além disso, são introduzidas algumas ferramentas existentes para a monitorização e controlo energético de **GPU**s. Por fim, são expostos alguns trabalhos que, de alguma forma, se relacionam com o trabalho desenvolvido.

2.1 Enquadramento

Para que seja possível compreender o âmbito do problema que esta dissertação se propõe a resolver, é necessário, primeiramente, estar na posse de alguns conhecimentos básicos sobre **Aprendizagem Profunda**, nomeadamente sobre o que compõe um modelo, como se processa o treino de um modelo e, ainda, como é que esta orquestração de tarefas é definida.

Por outro lado, as ferramentas de controlo e monitorização energética de **GPU**s atuais apresentam funcionalidades distintas e é importante compreender o funcionamento e as particularidades destas (p.ex., compatibilidades com distribuidoras de **GPU**s) que irão ajudar a definir qual a melhor opção para este projeto.

2.1.1 Aprendizagem Profunda

Aprendizagem Profunda é uma área bastante vasta da **Aprendizagem Máquina (AM)**. Sendo assim, nesta dissertação, o nosso foco serão as **Redes Neuronais Artificiais (RNA)**, em particular as **Redes Neuronais Convolucionais (RNC)**, que são modelos constituídos por várias camadas de unidades computacionais que se conectam entre si e têm capacidade de aprendizagem. Tal como o nome indica, estes modelos visam imitar o comportamento do cérebro humano, por isso, tais unidades computacionais são frequentemente denominadas de neurónios [10].

Redes Neuronais Artificiais

As **Redes Neuronais Artificiais** são compostas por neurónios que, por sua vez, consistem em [10]:

- Um conjunto de conexões a outros neurónios, através das quais recebem estímulos como entrada, sendo o estímulo enviado pelo nodo j representado como x_j. Cada uma dessas conexões tem um peso associado, p_{ij} representa o peso do estímulo enviado do neurónio j para o neurónio i.
- Um integrador, que transforma os n valores de entrada num só valor (p_i). Tipicamente é feito o somatório tendo em conta o peso de cada conexão.
- Uma função de ativação (f_a) utilizada para calcular o valor de saída do neurónio (s_i).

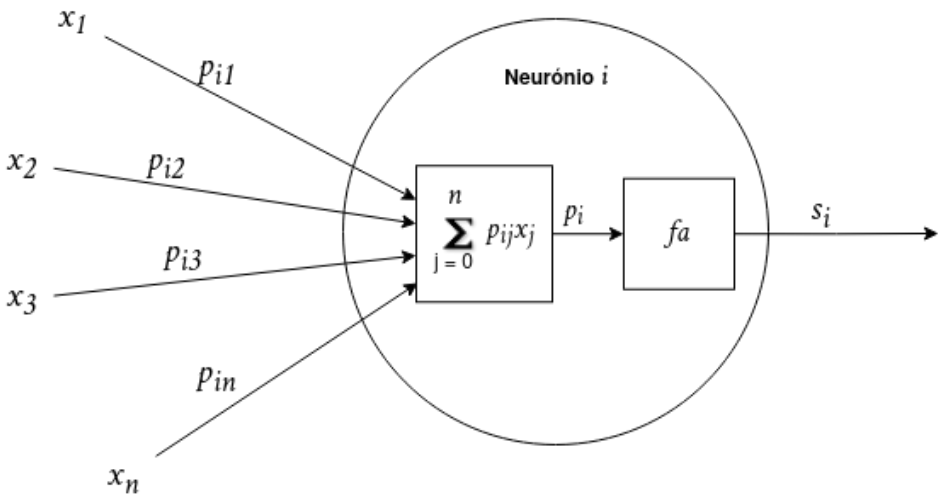


Figura 1: Composição de um neurónio artificial (adaptado de [10]).

A forma como os neurónios se conectam para formar a **RNA** denomina-se arquitetura (ou topologia) do modelo. Nestas arquiteturas existem três tipos de camadas [10]:

- A camada de entrada, que é a primeira camada da rede, usa os dados de entrada como estímulo inicial da rede e propaga-os para a camada seguinte, sem realizar computações.
- As camadas intermédias que são todas as camadas existentes entre a camada de entrada e a de saída. Estas camadas recebem estímulos de outras camadas (de entrada ou intermédias), aplicam a computação necessária e propagam o resultado para as seguintes camadas. A existência de camadas intermédias permite uma modelação de funções mais complexas, no entanto, provoca um aumento no tempo de treino.

- A camada de saída que recebe estímulos de outras camadas e gera o valor de saída da rede.

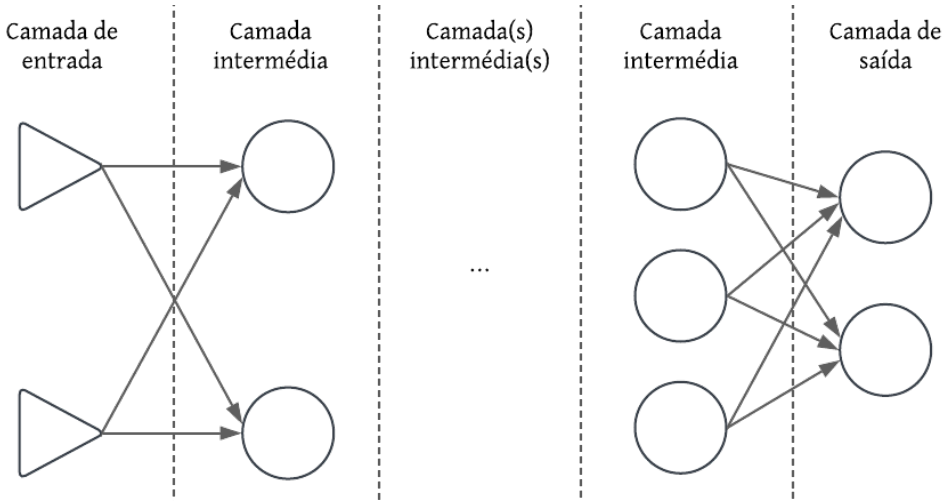


Figura 2: Camadas de uma **Redes Neuronais Artificiais**.

Redes Neuronais Convolucionais

As **Redes Neuronais Convolucionais** são um tipo de **RNA**s projetadas para reconhecer padrões e objetos em imagens e vídeos. O treino deste tipo de redes é feito utilizando um *dataset* de grande dimensão que contém imagens etiquetadas com informação sobre os objetos de interesse nelas representados. Durante o treino, a **RNC** aprende a associar características que reconhece nas imagens com as etiquetas disponíveis. Assim, após o treino, a rede deve ser capaz de escolher a etiqueta mais adequada para imagens que lhe são desconhecidas, isto é, que não foram utilizadas no treino [18].

Estas redes são compostas por camadas de três tipos: camadas convolucionais, camadas de *pooling* e camadas totalmente conectadas. A arquitetura típica destas redes, representada na Figura 3, consiste em repetições de conjuntos de camadas convolucionais seguidas de uma camada de *pooling* e terminando com uma ou mais camadas totalmente conectadas [46].

As camadas convolucionais são a unidade base de uma **RNC** e aplicam convoluções à imagem. Estas operações consistem na aplicação de filtros às várias partes de uma imagem, sendo que estes filtros são capazes de extrair diferentes características de cada região da imagem, dando origem a um mapa de características. As primeiras camadas extraem características mais simples e irão transmitir essa informação à camada seguinte. Assim, estas camadas são capazes de extrair características cada vez mais complexas ao longo da rede. Ainda, o facto dos filtros serem aplicados às várias partes de uma imagem faz com que estas redes sejam capazes de reconhecer padrões em qualquer parte de uma imagem [18].

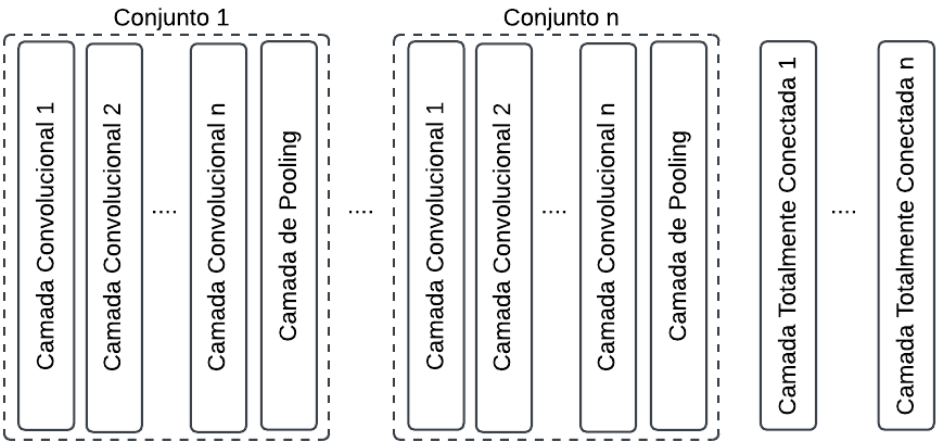


Figura 3: Arquitetura típica de uma **RNC** (baseado em [46]).

Por sua vez, uma camada de *pooling* é usada para diminuir a dimensão dos mapas de características produzidos pelas camadas convolucionais de forma a diminuir a complexidade computacional da rede. Estas camadas tentam manter as características que consideram essenciais e mais relevantes para a decisão final [18].

Finalmente, a camada totalmente conectada, tal como o nome indica, conecta todos os neurónios da camada anterior à camada atual. É uma camada de saída típica de **RNA**s que produz o resultado final, isto é, calcula a probabilidade da imagem estar associada a cada uma das etiquetas disponíveis e escolhe a etiqueta com maior probabilidade [18].

Treino de Modelos de AP

O treino dos modelos de **AP** divide-se em épocas constituídas por várias iterações, em que cada iteração consiste em utilizar como estímulo de entrada da rede um fragmento do *dataset* de treino, sendo que, no final de uma época, todos os fragmentos do *dataset* foram usados para alimentar a rede. No entanto, uma iteração começa bem antes das computações associadas à rede [25].

Os fragmentos do *dataset* utilizado estão, inicialmente, em armazenamento que pode ser local ou remoto (representado na Figura 4 através do número ⓿) e, como tal, a primeira fase de uma iteração é a fase de carregamento. Esta fase, representada na Figura 4 com o número ❶, consiste em carregar o fragmento do dispositivo de armazenamento para a memória principal (da **CPU**) [25].

De seguida, o fragmento passa, opcionalmente, pela fase de pré-processamento (identificada na Figura 4 com o número ❷). Esta fase pode consistir em vários passos para pré-processar os dados contidos no fragmento, como, por exemplo, no caso de imagens, a aplicação de filtros (corte, rotação, etc.)

que contribuam para um melhor desempenho do modelo. Esta fase, tipicamente, é da responsabilidade da **CPU** [25].

A fase seguinte, fase de cópia, consiste, tal como o nome indica, na cópia do fragmento da memória da **CPU** para a memória da **GPU** (identificada na Figura 4 com o número 3).

Apenas quando o fragmento percorreu todas as fases anteriormente descritas é que é possível utilizá-lo como estímulo da rede, na fase que denominamos como fase de computação (representada como número 4 na Figura 4). O algoritmo mais utilizado para o treino é o algoritmo de retropropagação, que consiste em duas fases. Inicialmente, dá-se a propagação, fase em que os estímulos são propagados e computados desde a camada de entrada até à camada de saída. De seguida, com base no resultado obtido na camada de saída, é calculada a distância desse valor em relação ao resultado esperado. Na segunda fase, a fase de retropropagação, a diferença obtida é propagada desde os neurónios da camada de saída até aos da camada de entrada para que calculem o gradiente, isto é, a taxa com a qual cada neurónio contribuiu para o erro. Este gradiente é utilizado para ajustar as equações associadas a cada neurónio de modo a que, na iteração seguinte, o resultado obtido esteja mais próximo do esperado [10].

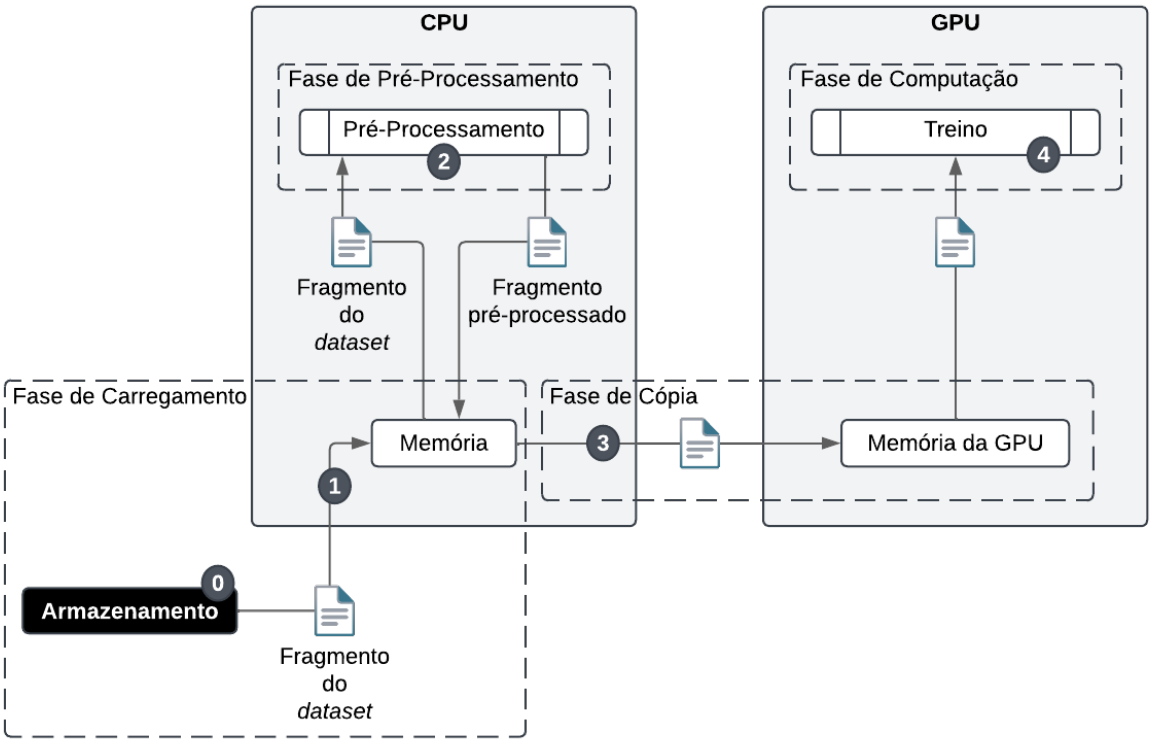


Figura 4: Fases de uma iteração do treino de modelos de **AP**.

Existem algoritmos que avaliam o desempenho do treino de um modelo e ajustam configurações de forma a obter melhores resultados, como o *train-test-validation split* que implica a divisão do *dataset* utilizado em três partes: uma parte, tipicamente maior, para treino, outra parte para testar o desempenho do modelo e, finalmente, outra parte para validar que o treino está a tomar a direção correta (isto é, garantir que o modelo têm bom desempenho não apenas para os dados usados para o treino mas também para dados diferentes desses - *overfitting*). Assim, no final de cada época, utilizando o *dataset* de validação que contém dados novos em relação ao *dataset* de treino, é estimado o erro. Com base nestas medidas, os hiperparâmetros e a complexidade do modelo são ajustados de forma a melhorar o seu desempenho e a generalizar melhor para casos que lhe são desconhecidos. No final do treino, é possível utilizar o *dataset* de teste, que contém dados aos quais o modelo nunca teve acesso, para avaliar a qualidade do modelo através de métricas como a acurácia, precisão, etc [6].

A paragem do processo de treino pode acontecer quando o modelo atinge a qualidade desejada (por exemplo, atinge uma determinada acurácia) ou quando é alcançado um número pré-definido de épocas [7].

Treino multi-GPU

No caso do treino multi-**GPU** é utilizada mais que uma **GPU** no treino do modelo, mas existem diferentes técnicas para distribuir o trabalho entre elas. Uma das técnicas mais utilizadas é o paralelismo de dados em que cada **GPU** mantém uma réplica do modelo para que cada uma delas aplique, simultaneamente, computações sobre um fragmento do *dataset* diferente. Também neste cenário existem as fases anteriormente descritas, sendo apenas acrescentada uma nova fase de sincronização antes do final da iteração, em que as **GPU**s partilham os gradientes calculados e, após terem recebido os gradientes de todas as outras **GPU**s, utilizam-nos para fazer a atualização dos parâmetros da rede de forma a manterem as suas réplicas do modelo consistentes [17].

Esta fase de sincronização pode acontecer simultaneamente à fase da computação associada à retropropagação. A Figura 5.3 representa a fase de sincronização, assim como os diferentes processos associados à fase de computação de uma iteração num cenário de treino multi-**GPU**. As **GPU**s vão enviando os gradientes para as outras **GPU**s (sincronização na Figura 5.3) enquanto procedem com o cálculo dos gradientes das camadas seguintes (retropropagação na Figura 5.3). No entanto, as **GPU**s que consigam terminar a retropropagação mais cedo (**GPU**s 0, 1 e 3 na Figura 5.3) ficam inutilizadas até que a mais lenta (**GPU** 2 na Figura 5.3) termine e envie os gradientes calculados (representado pelo momento T na Figura 5.3), pois devem receber os gradientes calculados por todas as outras **GPU**s na fase de re-

tropropagação para poder procederem à atualização dos parâmetros do modelo (atualização do modelo na Figura 5.3) [25].

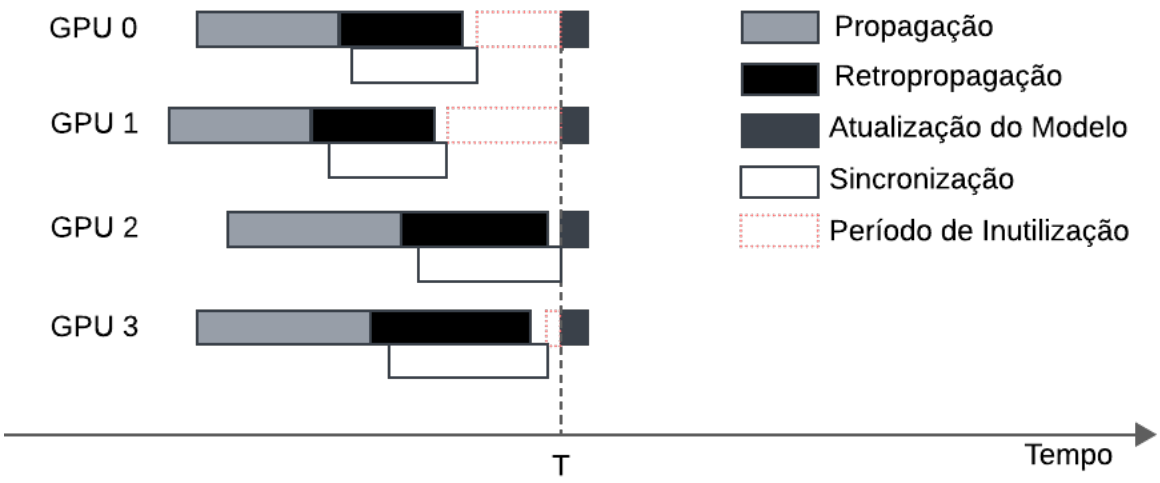


Figura 5: Sincronização em cenários de treino multi-**GPU**.

Assim, também a necessidade de sincronização e comunicação entre as várias **GPU**s envolvidas no treino pode levar a períodos de inutilização destes recursos. Ainda, esta fase, tal como a fase de computação, pode sobrepor-se com a fase de carregamento do fragmento do *dataset* a utilizar na iteração seguinte.

Classificação de Modelos

Um modelo pode ser intensivo em computação ou em movimento de dados, sendo a sua tipologia definida de acordo com a duração de cada uma das fases de uma iteração anteriormente descritas.

Por simplicidade, nesta secção iremos assumir apenas três fases: carregamento, cópia e computação. Consideramos daqui em diante que a fase de pré-processamento está integrada na fase de carregamento e, no caso de treino multi-**GPU**, a sincronização está integrada na fase de computação.

Apesar das fases de uma iteração, anteriormente descritas, acontecerem obrigatoriamente de forma sequencial, as diferentes iterações do treino de um modelo podem sobrepor-se parcialmente, já que cada uma das diferentes fases da iteração envolve a utilização de um conjunto diferente de recursos. Ainda, podemos considerar que existem duas *pipelines*: uma não dependente da **GPU**, que consiste na fase de carregamento dos dados e outra, dependente da **GPU**, que consiste na cópia dos dados para a memória deste recurso e a computação sobre os mesmos.

Estas duas *pipelines* operam paralelamente e, idealmente, a primeira *pipeline* seria rápida o suficiente para garantir que a **GPU** tivesse sempre dados disponíveis sobre os quais computar. Isto é, após terminar a computação sobre um fragmento (fase identificada com o número ③ na Figura 6 e que termina a T_{n+1}), o fragmento seguinte já estaria disponível para ser copiado para a memória da **GPU** (passo identificado com o número ④ na Figura 6). Quando isto se verifica, a primeira fase, de carregamento de um fragmento (representada através do número ① na Figura 6 e que termina também a T_{n+1}), sobrepõe-se totalmente com a execução da segunda *pipeline* sobre o fragmento anterior (fases identificadas pelos números ② e ③ na Figura 6). Um modelo em que tal suceda diz-se intensivo em computação e a **GPU** não terá períodos de inutilização por espera de dados [25].

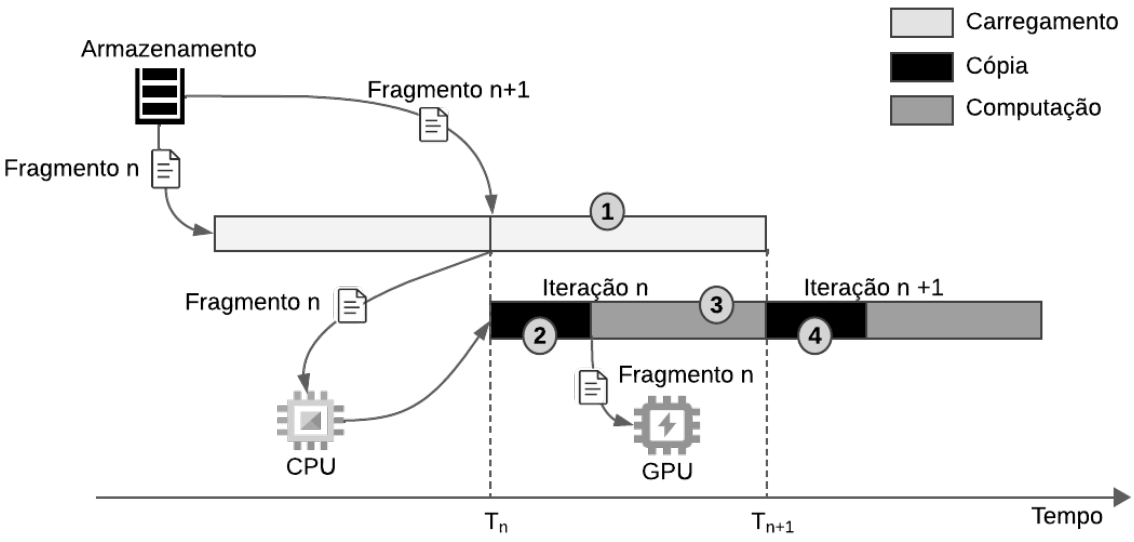


Figura 6: Exemplos da duração das fases da iteração num modelo intensivo em computação.

No entanto, é bastante comum que a primeira *pipeline* seja mais lenta, levando a períodos de inutilização da **GPU** (representado na Figura 7 entre os momentos T_n e T_{n+1}). Nestes casos, a **GPU** termina a computação sobre um fragmento (fase identificada pelo número ② na Figura 7) sem que o fragmento seguinte esteja disponível (i.e., sem que a fase identificada pelo número ① na Figura 7 termine), tendo de esperar até que tal se verifique (momento representado como T_{n+1} na Figura 7), adiando, assim, o início da fase de cópia do fragmento seguinte (representada como número ③ na Figura 7). Os modelos em que se identifica este comportamento dizem-se intensivos em movimento de dados [25].

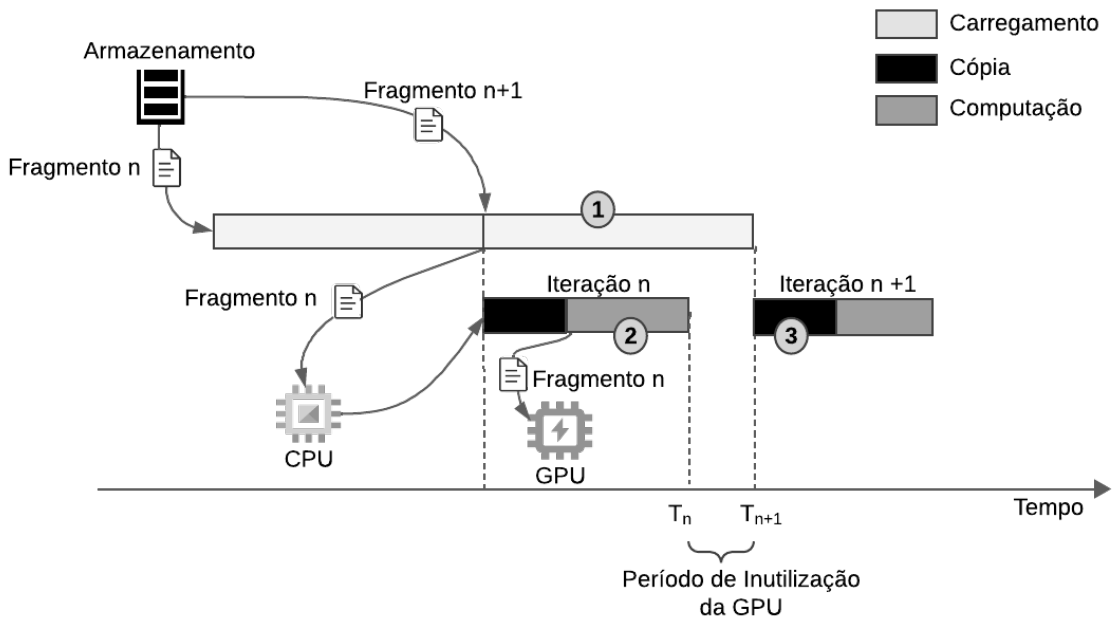


Figura 7: Exemplo da duração das fases da iteração num modelo intensivo em movimento de dados.

Gargalos de Desempenho

A classificação de um modelo depende da disponibilidade e desempenho dos recursos utilizados, já que estes serão fatores determinantes no que diz respeito à duração de cada uma das fases de uma iteração. Ainda, como o treino de modelos de **AP** utiliza todos os recursos num servidor, qualquer um deles pode ser o gargalo do treino, o que se refletirá na duração das fases das iterações. Para a entrada de dados é utilizado o dispositivo de armazenamento, a **RAM** e a **CPU**, já para a aplicação das computações sobre esses dados é utilizada a **GPU** [25]. Para além disso, quando o treino ocorre num ambiente distribuído ou quando o armazenamento é remoto, é também utilizada a rede [39].

Quando a largura de banda do dispositivo de armazenamento ou da rede, no caso do armazenamento ser remoto, está saturada, as leituras são lentas, não conseguindo acompanhar a velocidade de processamento da **CPU** e **GPU** [20]. Isto pode levar à inutilização da **CPU** e **GPU** durante os períodos de espera por dados, demonstrando que a rede ou o dispositivo de armazenamento podem tornar-se gargalos de desempenho. O mesmo pode suceder quando o espaço em memória é escasso: os fragmentos do *dataset* que podem ser acomodados no espaço disponível são pré-processados e copiados para a memória da **GPU** mais rapidamente do que são carregados novos fragmentos para a memória principal, deixando a **CPU** e a **GPU** sem dados para pré-processar [25]. Estes são cenários bastante frequentes devido ao crescimento dos *datasets* e do poder de processamento da **CPU** e da **GPU** que não tem sido

acompanhado por um crescimento proporcional do desempenho dos dispositivos de armazenamento. Os efeitos destes gargalos são visíveis na duração da fase de carregamento.

Outra possibilidade é o gargalo ser a **CPU**: o pré-processamento dos dados pode ser um passo bastante demorado, o que irá fazer com que a **GPU** espere por dados [26]. Também esta situação é recorrente tendo em conta que o crescimento do poder de processamento das **GPU**s tem sido bastante mais expressivo quando comparado com o crescimento do poder de processamento das **CPU**s (Lei de Huang - Figura 8). Quando tal gargalo se verifica, os seus efeitos são notórios na fase de pré-processamento.

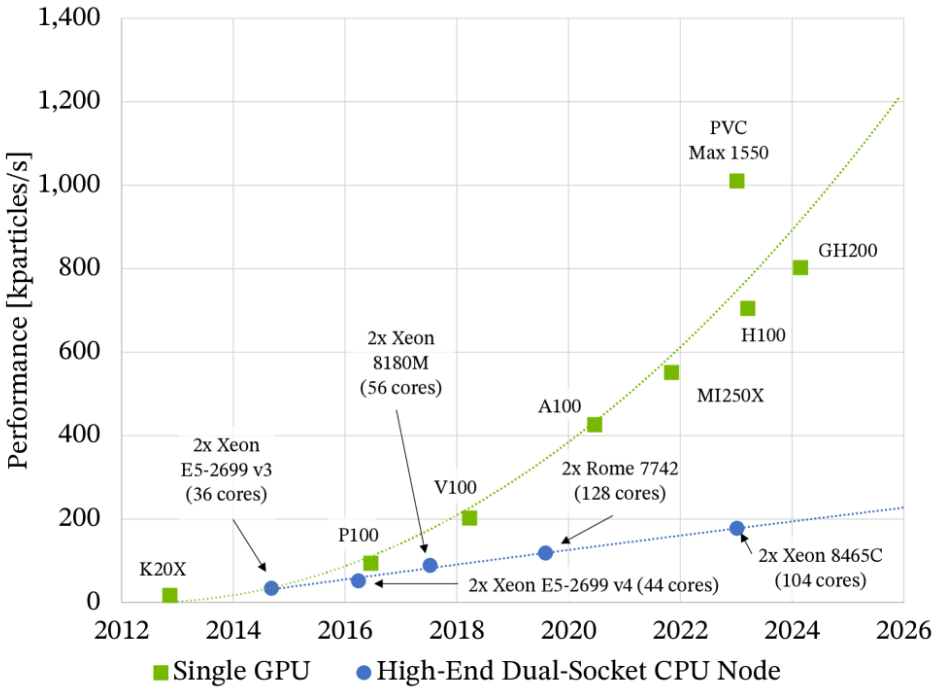


Figura 8: Poder de processamento de **CPU**s e **GPU**s ao longo dos anos [40].

Algumas soluções atuais tentam combater este gargalo como é o caso da biblioteca NVIDIA Developer Data Loading Library (DALI) [28]. Esta biblioteca torna possível fazer o pré-processamento de dados na **GPU** tornando este processo mais rápido. Existe ainda o Fast Forward Computer Vision (FFCV) [21] que analisa a *pipeline* de processamento de dados fornecida pelo utilizador e automaticamente compila-a para código máquina otimizado. Para além disso, faz ainda uma gestão otimizada da memória, evitando alocação desnecessária. Estas soluções procuram otimizar a fase de pré-processamento, diminuindo o seu tempo de execução.

Por outro lado, o gargalo de desempenho também pode estar relacionado com a fase de cópia dos fragmentos da memória principal para a memória da **GPU**. Tal sucede devido ao espaço da memória da **GPU** ser limitado: este recurso deve acomodar o modelo e respetivos parâmetros, sendo o espaço res-

tante utilizado para armazenar fragmentos do *dataset*. Assim, pode não restar muito espaço na memória da **GPU**, o que torna possível que o treino dos fragmentos que podem ser acomodados seja mais rápido do que a reposição com novos fragmentos.

O gargalo pode, ainda, ser a **GPU**, tal acontece quando a fase de computação é o que ocupa a grande parte do tempo da iteração. Ferramentas como a biblioteca NVIDIA CUDA Deep Neural Network (cuDNN) [27] pretendem melhorar este gargalo, melhorando o desempenho das operações intensivas em computação. Por sua vez, o MetaFlow [16] aproveita o facto de uma **RNA** poder ser traduzida num grafo computacional de operações matemáticas uma vez que é composta por unidades computacionais (neurónios) que aplicam cálculos e estão conectados entre si. Assim, dado um grafo computacional, este sistema encontra um outro equivalente, mas que permita alcançar um melhor desempenho no que diz respeito ao tempo de treino.

Por fim, quando o treino ocorre em mais que uma **GPU** ou até em mais que um nodo computacional, o gargalo pode ser a comunicação e sincronização entre os vários dispositivos. Independentemente da técnica escolhida para distribuir o trabalho (p.ex., paralelismo de dados, paralelismo de *pipeline*), é sempre necessário que os participantes do treino sincronizem, seja para garantir o uso dos mesmos parâmetros do modelo ou devido à dependência de dados. Estas particularidades do treino multi-**GPU** ou distribuído aumentam o tempo de execução e podem ainda levar à inutilização das **GPU**s pois podem ter de esperar pelos momentos de sincronização. As ferramentas atuais, utilizadas nestes casos, focam-se na otimização das primitivas utilizadas para a comunicação como é o caso da biblioteca NVIDIA Collective Communications Library (NCCL) [29] , ou ainda tentam planear as comunicações de forma a sobrepô-las com outras tarefas como o GradientFlow [39] e o TicTac [14]. Todas estas ferramentas foram propostas com o intuito de diminuir o impacto das comunicações/sincronizações no tempo de execução.

2.1.2 Controlo Energético de GPUs

Com o aumento da preocupação com o consumo energético das **GPU**s, os seus fornecedores desenvolveram *software* capaz de monitorizar e ajustar a frequência e/ou a potência de forma a proporcionar um uso mais eficiente a nível energético dos seus dispositivos. Nesta dissertação, o foco serão **GPU**s dedicadas, sendo assim, esta subsecção apresenta algumas ferramentas para esse tipo de dispositivos.

NVIDIA

No contexto de controlo energético de **GPU**s, a NVIDIA fornece a biblioteca NVIDIA Management Library (NVML) [30], cujo código não está disponível publicamente. Através desta biblioteca é possível monito-

rizar algumas métricas relativas a **GPU**s NVIDIA como consumo energético, potência e temperatura. A biblioteca permite ainda monitorizar a utilização e frequência da **GPU** e da memória da **GPU**.

Para além das funcionalidades de monitorização, é ainda possível impor limites no que diz respeito à potência da **GPU** e à frequência dos **Streaming Multiprocessor (SM)** e da memória da **GPU**, o que produz efeitos diretos no consumo energético.

A alteração da potência máxima pode ser feita para qualquer valor entre a potência máxima e mínima suportada pelo dispositivo, valores que podem ser consultados também através da biblioteca. Por outro lado, é ainda possível consultar os valores de frequência suportados para a memória da **GPU** e definir o valor máximo e mínimo pretendido tendo esses valores em conta. Para cada valor de frequência de memória da **GPU**, existe um conjunto de valores de frequência possíveis para a **GPU** (**SM**s). Assim, tal como para a memória da **GPU**, pode manipular-se a frequência da **GPU** definindo um valor máximo e mínimo de acordo com os valores suportados.

AMD

A AMD criou a Radeon Open Compute (ROCm) [4] que é uma pilha de *software* onde inclui duas bibliotecas através das quais é possível monitorizar métricas relevantes das **GPU**s AMD. Essas bibliotecas são a Radeon Open Compute System Management Interface (ROCm SMI) [5] e a Advanced Micro Devices System Management Interface (AMD SMI) [3], sendo que a AMD SMI é a sucessora da ROCm SMI.

As duas bibliotecas têm o respetivo código disponível publicamente e disponibilizam funcionalidades bastante semelhantes. Ambas permitem monitorizar a potência, a frequência da memória e dos **SM**s, o consumo energético e a utilização das **GPU**s, para além disso, é ainda possível alterar os limites de potência e de frequência.

A potência pode ser manipulada definindo um novo limite máximo ou então utilizando um dos perfis de potência disponibilizados como o perfil de poupança de potência que adota uma potência mais baixa do que a padrão. Por outro lado, a frequência máxima e mínima também podem ser alteradas, mas apenas para certos valores dentro dos limites da máquina, que podem ser consultados recorrendo à ferramenta.

Independente de Hardware

Existe ainda uma ferramenta de monitorização que recorre ao uso de várias bibliotecas por forma a ser capaz de controlar e monitorizar o consumo energético tanto de **GPU**s da NVIDIA como da AMD: a biblioteca Performance Application Programming Interface (PAPI) [1]. O código desta ferramenta está

disponível publicamente e recorre à NVML [30] e à ROCm SMI [5] de forma a disponibilizar métricas sobre **GPU**s da NVIDIA e AMD, respetivamente.

A biblioteca PAPI utiliza a NVML para monitorizar a frequência, a potência, a temperatura e a utilização da **GPU**. No entanto, até ao momento, suporta apenas a imposição de limites à potência e não à frequência, apesar de ser possível fazê-lo através da NVML. Através da biblioteca ROCm SMI, monitoriza e limita a potência e a frequência das **GPU**s AMD.

Técnicas de Otimização do Consumo Energético

Tanto as **GPU**s como a respetiva memória, atuam, por padrão, à frequência máxima [13], no entanto, utilizando ferramentas como as apresentadas anteriormente, a frequência a que operam pode ser ajustada. Este ajuste tem impacto direto no tempo de execução da **GPU**, mas também no seu consumo energético.

No entanto, no que diz respeito à frequência da memória da **GPU**, está comprovado que a frequência máxima é a configuração que permite alcançar melhor desempenho e menor consumo energético [24]. Ainda, o consumo energético desta componente representa uma parte negligenciável do consumo energético total da **GPU**, pelo que, em grande parte dos sistemas de otimização do consumo energético, a sua frequência de atuação é mantida no valor máximo [24].

Por outro lado, a relação entre a frequência da **GPU** e estas métricas é mais complexa [43]. Ao utilizar a frequência máxima do dispositivo é alcançado o menor tempo de execução possível a custo de um alto consumo energético, porém, isto não significa que utilizando a frequência mínima seria consumida menos energia. Nesse cenário, a **GPU** tornar-se-ia muito lenta, tornando o tempo de execução significativamente maior. Assim, apesar de consumir menos Joules por unidade de tempo, acabaria por consumir mais energia para completar a tarefa devido ao aumento expressivo do tempo de execução.

A manipulação da potência e da frequência dos processadores denomina-se de **Dynamic Voltage and Frequency Scaling (DVFS)**. Esta técnica, para além de ser utilizada no contexto de otimização de consumo energético, é também adotada para melhorar a alocação de recursos. Existe ainda uma técnica denominada **Dynamic Resource Sleep (DRS)** que consiste em desligar servidores quando inutilizados como forma de diminuir o consumo energético, pois, mesmo sem tarefas para processar, consomem energia, derivada da potência estática - potência gasta sem qualquer tarefa. No entanto, na utilização desta prática, é necessário ter em conta que, se for necessário ligar novamente o servidor, haverá um consumo energético extra. Adicionalmente, enquanto o servidor é ligado, podemos estar em incumprimento no que diz respeito à qualidade de serviço das aplicações. Assim, entre deixar um recurso inutilizado por algum tempo ou desligá-lo, poderá ser vantajoso deixá-lo ligado para não incorrer no custo

extra envolvido em ligá-lo de novo [43].

2.2 Trabalho Relacionado

Nesta secção são analisados alguns trabalhos relacionados com o problema que esta dissertação pretende resolver de forma a evidenciar as limitações destas soluções e, consequentemente, a necessidade de uma nova solução. São mencionados trabalhos com foco na otimização da eficiência energética no contexto de treino de modelos de **AP**, assim como sistemas para provisionamento de potência em infraestruturas utilizadas no mesmo âmbito e, por fim, soluções para otimização da eficiência energética em contextos mais genéricos.

2.2.1 Otimização da Eficiência Energética do Treino de Modelos de AP

Esta subsecção apresenta alguns sistemas que têm como objetivo melhorar a eficiência energética do treino de modelos de **AP**, isto é, diminuir o consumo energético sem grande prejuízo para o tempo de execução. Para tal, foram propostas ferramentas cujo foco é minimizar períodos de inutilização da **GPU** através da manipulação da sua frequência [8, 9] ou que recorrem a modelos preditivos para a escolha da frequência ótima [41, 49]. Outras soluções recorrem ainda à alteração do limite de potência e de hiperparâmetros do treino [47].

O EnvPipe [8] e o Perseus [9] são ferramentas que exploram a minimização dos períodos de inutilização da **GPU** como forma de melhorar a eficiência energética. Ambos os sistemas foram desenhados para ambientes de treino multi-**GPU** que é aplicado paralelismo de *pipeline* uma vez que a utilização desta técnica implica a existência de bolhas na *pipeline*, isto é, períodos de inutilização da **GPU**. Estas bolhas surgem quando uma **GPU** termina uma tarefa mas não pode dar início à tarefa seguinte devido à dependência de dados, ficando inutilizada até que a outra **GPU** lhe envie os dados de que precisa para dar início à tarefa seguinte. Estes sistemas identificam as bolhas existentes e verificam se é possível estender a tarefa anterior de forma a ocupar o tempo da bolha sem que isso leve a que uma outra **GPU** fique à espera de dados e atrase toda a *pipeline*. Nos casos em que tal acontece, os sistemas ajustam a frequência de forma a que as bolhas sejam minimizadas sem que o tempo de execução global do treino seja prejudicado, diminuindo assim o consumo energético do treino.

Adicionalmente, o EnvPipe faz o agendamento das unidades de *pipeline* de forma a maximizar o número de bolhas com potencial para serem estendidas sem quebrar dependências de dados nem degradar o desempenho. Este sistema define o plano para o agendamento das unidades que compõe o

pipeline, assim como o plano de frequência para cada uma delas. Estes planos são aplicados durante todo o treino, não se adaptando à possível variabilidade na disponibilidade e desempenho dos recursos utilizados no treino dos modelos.

Por sua vez, o Perseus explora ainda períodos de inutilização derivados do uso de vários nodos para o treino (treino distribuído). Nesses casos, existem várias réplicas da mesma *pipeline* a correr em paralelo que devem sincronizar no final de cada iteração. Assim, diferentes *pipelines* correm a velocidades diferentes, pelo que todas as *pipelines* que acabam a iteração mais cedo do que a *pipeline* mais lenta estão a ser desnecessariamente rápidas e, consequentemente, estão a gastar energia ineficientemente. Desta forma, o Perseus atrasa essas *pipelines* sem que isso atrase a sincronização dos gradientes e, consequentemente, o tempo de execução global. Tal como, o EnvPipe, o Perseus também define o plano de frequência para cada unidade do *pipeline*, alterando este plano apenas se a infraestrutura de *hardware* (p.ex., gestor de potência do bastidor do *data center*) o notificar da existência de um gargalo e informar sobre o atraso que será causado.

Outro género de sistemas que têm como objetivo otimizar o consumo energético no contexto de **AP** são os que recorrem a modelos preditivos como é o caso do GPOEO [41] e do DVFO [49].

A operação do GPOEO [41] divide-se numa fase *offline* e numa fase *online*. Na fase *offline* executa tarefas de referência de forma a obter dados que relacionem diferentes configurações de frequência com o tempo de execução e energia consumida. Esses dados são utilizados para alimentar um modelo que, posteriormente, é utilizado para ajudar a encontrar a frequência ótima para o treino de modelos. Assim, na fase *online*, quando é submetida uma tarefa no sistema, esse modelo é utilizado para prever o tempo de execução da mesma e a energia consumida para cada um dos valores de frequência disponíveis. Tendo em conta essas previsões, o GPOEO escolhe como frequência ótima aquela que permite um menor consumo energético sem degradar o tempo de execução em mais de 5%. Após aplicar essa frequência, monitoriza o treino e, caso não estejam a ser cumpridos os requisitos de tempo de execução, é reiniciada a fase *online*.

O modelo treinado durante a fase *offline* deste algoritmo não garante que seja encontrada a frequência ótima uma vez que diferentes modelos de **AP** têm diferentes comportamentos perante a mesma configuração de frequência. Assim, o comportamento registado pelas tarefas de referência na fase *offline* pode ser completamente distinto do comportamento do modelo a treinar. Adicionalmente, a disponibilidade dos recursos durante a fase *offline* e *online* pode ser diferente e, sendo que este fator também influência o tempo de execução e consumo energético, as previsões do modelo perdem qualidade com estas diferenças. Deste modo, o modelo pode levar à escolha de frequências distantes da configuração ótima.

Por outro lado, o DVFO [49] é uma *framework* cujo foco é a inferência tendo em conta o paradigma *edge-cloud collaborative*. Neste paradigma, são carregados mapas parciais das variáveis da **RNA** dos dispositivos *edge* para os servidores em nuvem. Assim os dispositivos *edge* fazem inferência parcial, sendo o resto executado pelos servidores, e usam pequenas **RNA**s para fundir os resultados e obter, assim, o resultado final da inferência. Os utilizadores submetem tarefas de inferência ao DVFO acompanhadas de parâmetros que definem a importância dada ao consumo energético e à latência. O DVFO começa por extrair as variáveis dos dados de entrada e obter o mapa de variáveis da rede. De seguida, analisa a importância de cada variável para definir quais serão carregadas para o servidor em nuvem. Com base em execuções anteriores, na largura de banda da rede atual e nas configurações do utilizador, é utilizado um modelo que aprende o vetor de frequências ótimas para a **GPU**, **CPU** e memória do dispositivo *edge* assim como a proporção dos parâmetros do mapa de variáveis que deve ser carregada. Assim, com base nessa proporção, o sistema retém as variáveis com maior importância para dar início à inferência local utilizando as frequências ótimas encontradas. As restantes variáveis, de importância secundária, são carregadas para a nuvem. Os servidores em nuvem enviam o resultado da inferência remota que é fundido no dispositivo *edge* com o resultado local, sendo assim obtido o resultado final da inferência.

Por fim, o Zeus [47] manipula o *batch size* e o limite de potência para otimizar o consumo energético e o tempo de treino de acordo com a relevância dada pelo utilizador a cada um destes fatores. Os valores ótimos para o *batch size* e limite de potência são obtidos através da minimização de uma função de custo que têm em conta a importância dada pelo utilizador ao consumo energético. Adicionalmente, o Zeus assume que os modelos são treinados de forma recorrente e ajusta o seu comportamento a cada recorrência conforme o tempo de execução e consumo energético registado para a frequência escolhida.

Pelo facto de ajustar o *batch size*, que é um hiperparâmetro definido pelo utilizador, pode estar a alterar a eficiência estatística do modelo. Para além disso, este sistema foca-se em otimizar a eficiência energética apenas de treinos recorrentes.

2.2.2 Provisionamento de Potência numa Infraestrutura para Treino de Modelos de AP

Nesta subsecção são apresentados trabalhos que realizam provisionamento de potência no contexto de infraestruturas de treino de modelos de **AP**, isto é, distribuem a reserva de potência da infraestrutura pelos vários recursos que a constituem. Por um lado, foram desenvolvidas soluções que recorrem ao provisionamento de potência com o objetivo de acomodar mais servidores na infraestrutura [33] e, por outro lado, existem ferramentas que utilizam essa técnica para minimizar o tempo de execução das tarefas

a executar na infraestrutura [13].

O POLCA [33] enquadra-se no primeiro grupo de ferramentas e, para cumprir o objetivo de acomodar mais servidores na infraestrutura respeitando o seu limite de potência, diminui o pico de potência dos vários servidores. Para isso define uma política de dois limites de potência a nível da *row* (conjunto de máquinas na infraestrutura ligadas à mesma linha energética) e divide os trabalhos em dois níveis de prioridade (alta e baixa). Quando é atingido o limite de potência mais baixo da *row* é diminuída a frequência para os trabalhos de baixa prioridade. Por outro lado, quando é atingido o limite mais alto é diminuída ainda mais a frequência dos trabalhos de baixa prioridade e, caso mesmo assim o limite ainda não seja respeitado, é diminuída a frequência para trabalhos de alta prioridade.

Por sua vez, o PowerFlow [13] tem como objetivo diminuir o tempo de execução das tarefas respeitando o limite de energia da infraestrutura. Por forma a alcançar esse objetivo, começa por calcular índices de prioridade para cada trabalho com base nos ganhos obtidos se lhe for atribuída uma **GPU** adicional ou se for aumentada a frequência das **GPU**s em que está alocado. Com base nesses índices, aloca os trabalhos pelas **GPU**s disponíveis até que todas estejam alocadas ou até que o limite de potência da infraestrutura seja alcançado. Após esta alocação, se o limite ainda não foi alcançado, o sistema incrementa gradualmente a frequência das **GPU**s. Para além disso, usa técnicas para evitar que os trabalhos utilizem **GPU**s em vários nodos computacionais para evitar a fragmentação de recursos. Recorre também à técnica **DRS**, isto é, quando existem servidores inutilizados são desligados para não contribuírem para o aumento da potência da infraestrutura. Desta forma, sobra mais potência para os restantes servidores que realmente estão a realizar trabalho, podendo, assim, aumentar a frequência dos dispositivos nesses servidores e, consequentemente, diminuir o tempo de execução dos seus trabalhos.

2.2.3 Otimização de Eficiência Energética de GPUs

Nesta subsecção são apresentadas ferramentas utilizadas para otimizar a eficiência energética de **GPU**s em contexto genérico, isto é, para qualquer tipo de tarefas que executem em **GPU**s. Neste âmbito existem ferramentas que recorrem ao agendamento das tarefas e à técnica **DVFS** [43], ou apenas ao **DVFS** [48] ou ainda ao provisionamento de potência [31].

Wang et al. [43] desenvolveram um sistema capaz de fazer agendamento de tarefas de referência [42] *online* e *offline* com um prazo associado de forma eficiente em termos energéticos. Para isso desliga servidores que estão inutilizados durante algum tempo usando **DRS** (tendo em conta que se for necessário ligá-los de novo isso implicará um consumo extra de energia) e atribui as tarefas a pares **CPU**-**GPU** de forma a respeitar os seus prazos. Assim, organiza as tarefas pela proximidade do seu prazo e, para cada

uma delas, calcula o tempo de execução com a frequência ótima e tenta atribuí-la ao par **CPU**-**GPU** com menor carga. Caso não seja possível respeitar o prazo com o tempo de execução obtido, então é utilizada uma frequência mais alta (a frequência mais baixa que permita respeitar o prazo associado à tarefa). Caso mesmo assim não seja possível cumprir o prazo, então a tarefa é alocada a um par **CPU**-**GPU** anteriormente inutilizado ou num servidor desligado. Ao assumirem que uma tarefa só pode ser alocada a um único par **CPU**-**GPU**, descartam, assim, o uso deste sistema para treino multi-**GPU** de modelos de **AP**.

Por sua vez, o GEEPAFS [48] recorre apenas à técnica **DVFS**, sendo que o objetivo principal deste sistema é diminuir o consumo energético de **GPU**s assegurando, simultaneamente, que a perda de desempenho das aplicações afetadas é limitada. Para alcançar o objetivo, este sistema faz a modelação *online* do desempenho para cada configuração de frequência utilizando contadores de *hardware* da **GPU**. Os contadores considerados dizem respeito à utilização da largura de banda da memória das **GPU**s, métrica correlacionada com o desempenho da aplicação, segundo os autores. O modelo construído deverá prever o desempenho da aplicação para diferentes frequências de atuação da **GPU**. Assim, com base nessas previsões, o sistema altera a frequência da **GPU** de forma a melhorar a eficiência energética enquanto assegura que a perda de desempenho é limitada (de acordo com o objetivo do utilizador).

Por fim, o APS [31] atua a nível do servidor para distribuir a potência pelos dispositivos (**CPU**s, **GPU**s) de acordo com a sua utilização, de forma a respeitar um limite de potência definido pelo administrador ou um gestor de potência de alto nível, isto é, a nível do bastidor ou da infraestrutura, por exemplo. Para cumprir esse objetivo, começa por utilizar *stressmarks* para gerar, para cada dispositivo, uma tabela que associa cada configuração de frequência à potência máxima consumida. Tendo as tabelas, o sistema começa a monitorizar continuamente o consumo de potência e a utilização de todos os dispositivos. Depois de agregar várias amostras de consumo de potência e utilização, é despoletado o algoritmo que distribui a potência disponível ao nível da infraestrutura por todos os dispositivos com base na sua utilização. De seguida, de acordo com as tabelas geradas, é aplicada a frequência adequada a cada dispositivo. Se a potência total ultrapassar o limite que é definido pelo administrador é iniciada uma fase em que o sistema diminui de forma igual a potência de todos os dispositivos sem ter em conta a utilização. Isto porque, assim que o limite estiver a ser respeitado de novo, o algoritmo de distribuição irá corrigir essa situação de acordo com os novos valores de utilização.

2.2.4 Sumário

Existem já alguns sistemas que se propõem a melhorar o consumo energético de **GPU**s. No entanto, alguns deles restringem-se a cenários de treino multi-**GPU** e distribuído (em particular, quando é usada a técnica de distribuição de paralelismo de *pipeline*) [8, 9] ou a cenários de treino com uma única **GPU** [41, 43, 47]. Ainda, existem ferramentas cujo foco é uma fase específica da *pipeline* de **AP** (inferência - [33, 49]). Por outro lado, existem sistemas mais genéricos que prometem melhorar a eficiência energética de qualquer carga de trabalho que execute em **GPU** [31, 43, 48]. No entanto, o treino de modelos **AP** é composto por várias fases com diferentes níveis de carga para a **GPU**, assim, decisões tomadas sem atenção a essas características podem levar à utilização de frequências inadequadas ou sub-ótimas.

Para além disso, no conjunto de ferramentas apresentadas, existem sistemas que recorrem à alteração de hiperparâmetros definidos pelo utilizador para melhorar a eficiência energética, podendo prejudicar a eficiência estatística do modelo [47], e soluções que dependem de modelos preditivos [41, 49] e que, por isso, dependem das cargas de trabalho utilizadas para o treino desses modelos. Adicionalmente, algumas soluções propostas dependem da análise prévia do tempo de execução e consumo energético sobre diferentes configurações de frequência utilizando tarefas de referência [31, 41] ou até mesmo da execução prévia de parte da tarefa para recolher informação sobre o seu comportamento [13]. No entanto, o treino de diferentes modelos de **AP** tem diferentes sensibilidades à alteração da frequência, pelo que analisar o comportamento de outros modelos ou usar o seu comportamento para treinar um modelo preditivo pode levar à escolha de uma configuração de frequência distante da ótima. Ainda, estas análises e modelos não têm em conta a disponibilidade dos recursos em tempo de execução, que pode ser diferente do que se verifica no momento da análise.

A Tabela 1 descreve, para cada ferramenta, a área e cenários de treino (Treino Single-**GPU** - numa única **GPU**, Multi-**GPU** e Distribuído) em que a mesma é aplicável. São ainda identificadas as técnicas utilizadas por cada ferramenta e os recursos por estas afetados. Por fim, é indicado o tipo de *profiling* aplicado, para as ferramentas em que tal técnica é utilizada. O *profiling* consiste na análise do comportamento da aplicação para tomar decisões que pode ser *online* se for feita durante a execução da aplicação ou *offline* se acontecer à *priori*.

Resumidamente, a ferramenta desenvolvida avança o estado da arte por ser aplicável em cenários de treino em uma ou mais **GPU**s, analisar, em tempo de execução, as características da carga de trabalho e, com base nessa análise, utilizar apenas a manipulação da frequência (**DVFS**) para melhorar a eficiência energética do treino de modelos. O fator que mais diferencia o WAFT será a atenção às diferentes fases de uma iteração do treino de modelos **AP** já que cada uma delas envolve a utilização de diferentes recursos

a diferentes níveis. Em particular, os gargalos no carregamento de dados são identificados como uma oportunidade de otimização do consumo energético do treino. Ainda, o trabalho desenvolvido realça a necessidade de adaptação do algoritmo utilizado às características do modelo em questão.

Tabela 1: Resumo do trabalho relacionado.

Ferramenta	Área de Aplicação	Treino			Recurso(s) Afetado(s)	Técnica(s)	Profiling
		Single-GPU	Multi-GPU	Distribuído			
EnvPipe [8]	PP		X	X	G	●○	ON
Perseus [9]	PP		X	X	G	●○	ON
Zeus [47]	AP	X	X		G	●◑◒	N/A
GPOEO [41]	AP	X			G	●	OFF
DVFO [49]	INF				GCM	●◒	N/A
POLCA [33]	INF LLM	X	X	X	G	●◐	N/A
PowerFlow [13]	AP	X	X	X	G	◐◓○	ON + OFF
Wang et al. [43]	-	X			GMg	●○◓	N/A
APS [31]	-	X	X	X	GC	●◐	N/A
GEEPAFS [48]	-	X	X		G	●	ON
WAFT	AP	X	X		G	●	ON

Área de Aplicação:

PP Paralelismo de *Pipeline*

AP Aprendizagem Profunda

LLM Modelos de Linguagem em Larga Escala

– Genérico

Recurso(s) Afetado(s):

G GPU

C CPU

M Memória principal

Mg Memória da GPU

Técnica(s):

● **DVFS**

○ Agendamento

◑ Alteração de Hiperparâmetros

◒ Execuções Passadas

◐ Provisionamento de Potência

◓ **DRS**

Profiling

ON Online

OFF Offline

N/A Não Aplicável

Capítulo 3

Estudo Motivacional

Este capítulo apresenta um estudo motivacional cujo objetivo é compreender o funcionamento do treino de modelos de **AP** e os efeitos que podem surgir nesse contexto por meio da aplicação de diferentes configurações de frequência de atuação à **GPU**, isto é, alterações no tempo de iteração e, ainda, na duração das fases que a compõem.

3.1 Ambiente Experimental e Metodologia de Teste

Nesta secção são apresentados os ambientes em que foram realizados os testes, assim como a metodologia usada, cargas de trabalho e métricas recolhidas. De seguida, são ainda descritos os cenários de teste cujos resultados são expostos nas próximas secções.

3.1.1 Ambiente de Teste

Foram utilizados dois ambientes de teste diferentes:

- **Ambiente de Teste A: Commodity.** Uma máquina com uma **CPU** Intel Core i9-9900K e uma **GPU** NVIDIA GeForce RTX 2070 SUPER, sendo a memória da **GPU** de 8 GB. As frequências suportadas pela **GPU** variam entre 300 MHz e 2100 MHz com incrementos de 15 MHz entre uma frequência e a próxima (i.e., as frequências disponíveis entre o valor máximo e mínimo são múltiplos de 15), já a frequência da memória da **GPU** pode variar entre 405 MHz e 7001 MHz. A memória principal desta máquina é uma **Double Data Rate (DDR)** de 16 GiB. Esta máquina está ainda equipada com um disco **Non-Volatile Memory Express (NVMe)** de 1024 GB.

- **Ambiente de Teste B: Deucalion.** Uma máquina com duas **CPU**s AMD EPYC 7742 e quatro **GPU**s NVIDIA Ampere A100, sendo a memória destas de 80 GB. A frequência da memória das **GPU**s é de 1593 MHz já, no que diz respeito à frequência da **GPU**, esta pode variar entre 210

MHz e 1410 MHz, sendo cada frequência suportada separada da seguinte por 15 MHz. A memória principal desta máquina é uma **DDR** com capacidade para 512 GB. A máquina está equipada com um disco **Serial Advanced Technology Attachment (SATA)** de 480 GB tendo ainda acesso a armazenamento remoto através da rede recorrendo a dois adaptadores NVIDIA ConnectX-6 200 Gb/s.

3.1.2 Cenários de Teste

Por forma a responder às questões iniciais, foram definidos três cenários de teste:

- **Cópia Bloqueante vs. Não-Bloqueante.** Este cenário de teste consiste na comparação do desempenho do treino dos modelos com a cópia dos dados da memória principal para a memória da(s) **GPU**s em modo bloqueante e não bloqueante (i.e., quando se espera até a cópia estar concluída para prosseguir com a execução do treino em comparação com quando a cópia é feita assincronamente enquanto a execução do treino continua).
- **Limite de Frequência.** Neste cenário são aplicados limites fixos de frequência à **GPU** ao longo do treino para analisar o comportamento de cada um dos modelos.
- **Limite de Frequência por Fase.** Neste cenário de teste é analisado o impacto do uso de uma mesma frequência nas diferentes fases da iteração.

3.1.3 Cargas de Trabalho

Os resultados apresentados foram obtidos a partir do treino dos modelos AlexNet [19], ResNet101 e ResNet18 [15] com o *dataset* ImageNet [12] recorrendo à aplicação PyTorch [35] de acordo com as Tabelas 2 e 3, sendo que, no ambiente de teste *Commodity*, foi utilizado apenas cerca de 12% do *dataset*. O conjunto de modelos escolhidos foi definido de forma a incluir um modelo intensivo em movimento de dados (AlexNet), outro intensivo em computação (ResNet101) e, ainda, um que gasta, aproximadamente, tanto tempo no carregamento de dados como no processamento/computação sobre os mesmos (ResNet18).

O treino destes modelos divide-se em épocas, sendo que cada uma se caracteriza por duas fases: o treino e a validação do modelo. Como tal, foi necessário dividir o *dataset* em duas partes: uma parte maior para treino e outra para validação de acordo com as Tabelas 2 e 3. O critério de paragem utilizado nestes testes foi o número de épocas, fixado em três épocas.

Estas cargas de trabalho não inclui pré-processamento dos dados, pelo que esta fase não é representada neste estudo.

Tabela 2: Cargas de Trabalho - *Commodity*.

Modelo	Batch Size	Dataset
AlexNet	256	ImageNet (16GB / 3.8GB)
ResNet101	32	
ResNet18	128	

Tabela 3: Cargas de Trabalho - *Deucalion*.

Modelo	Batch Size	Dataset
AlexNet	256	ImageNet (146GB / 14GB)
ResNet101	256	

3.1.4 Métricas

Na realização destes testes foram monitorizados o consumo energético total relativo à(s) **GPU**(s) e o tempo de execução global do treino. Para além disso, ao longo do treino, foi registada a duração de cada uma das fases de cada iteração, assim como o consumo energético associado à execução da iteração.

3.1.5 Metodologia de teste

Estudos dizem que a frequência da memória das **GPU**s não tem uma influência significativa no consumo energético do sistema [43] e que a configuração de frequência da memória mais eficiente a nível energético é a frequência padrão (máxima) [24], por essas razões é apenas estudado o impacto do ajuste da frequência da **GPU**, que foi manipulada recorrendo à biblioteca NVML. Da mesma forma, o consumo energético envolvido nas leituras e escritas do dispositivo de armazenamento é considerado negligenciável em comparação com o consumo total do servidor, uma vez que, neste contexto, a **GPU** é sempre a responsável pela maior parte do consumo energético [13]. Por isso, o foco destes testes será o consumo energético das **GPU**s, sendo esta métrica recolhida também recorrendo à biblioteca NVML.

Por outro lado, no que diz respeito à média da duração de cada uma das fases da iteração, foi calculada através da recolha de *timestamps* aquando do fim de cada uma delas. Assim, a duração da fase de carregamento corresponde ao período de tempo entre a conclusão da computação sobre um fragmento até que o carregamento do fragmento seguinte está concluído. Isto significa que o tempo contabilizado para essa fase corresponde apenas à porção desta fase que não é sobreposta com a iteração anterior. Assim, a fase de carregamento nesta secção corresponde ao tempo em que a **GPU** está inutilizada à espera de dados e não ao carregamento completo do fragmento.

Os resultados apresentados a seguir foram obtidos calculando a média dos valores obtidos em três execuções do treino do modelo em questão e, em todas as execuções, foi garantido que o ambiente de teste apresentava as mesmas condições. No entanto, nos resultados do último cenário de teste (apresentados na secção 3.4), nomeadamente, nos que permitiram avaliar o impacto de diferentes frequências

nas diferentes fases de uma iteração, as métricas apresentadas correspondem a um excerto da execução do treino (100 iterações) para cada uma das configurações de frequência. Para esses testes foram considerados vários valores de frequência, nomeadamente o valor máximo (2100 MHz) e mínimo (300 MHz) suportados pela **GPU** da máquina onde o teste foi executado (*Commodity*).

3.2 Cópia Bloqueante vs. Não Bloqueante

O modo de cópia não bloqueante tem sido adotado para a transferência dos dados da memória principal para a memória da **GPU** pela ideia de que este modo será sempre melhor no que diz respeito ao tempo de execução do treino do modelo [22]. No entanto, a sua utilização retira-nos informação sobre o real momento em que a fase de cópia de dados termina. Deste modo, foi relevante avaliar se o facto de tornar a cópia bloqueante efetivamente tinha um impacto significativo no desempenho do treino dos modelos. Para tal, foram feitos testes sobre as duas diferentes configurações de cópia: bloqueante e não bloqueante; mantendo-se as configurações padrão de frequência (frequência máxima).

Commodity. Nesta máquina, fazendo a cópia dos fragmentos em modo não bloqueante, o treino do modelo AlexNet terminou após 20 minutos, consumindo um total de 87 kJ enquanto o treino do modelo ResNet101 demorou 58 minutos e foi registado um consumo energético de 663 kJ (Figura 9).

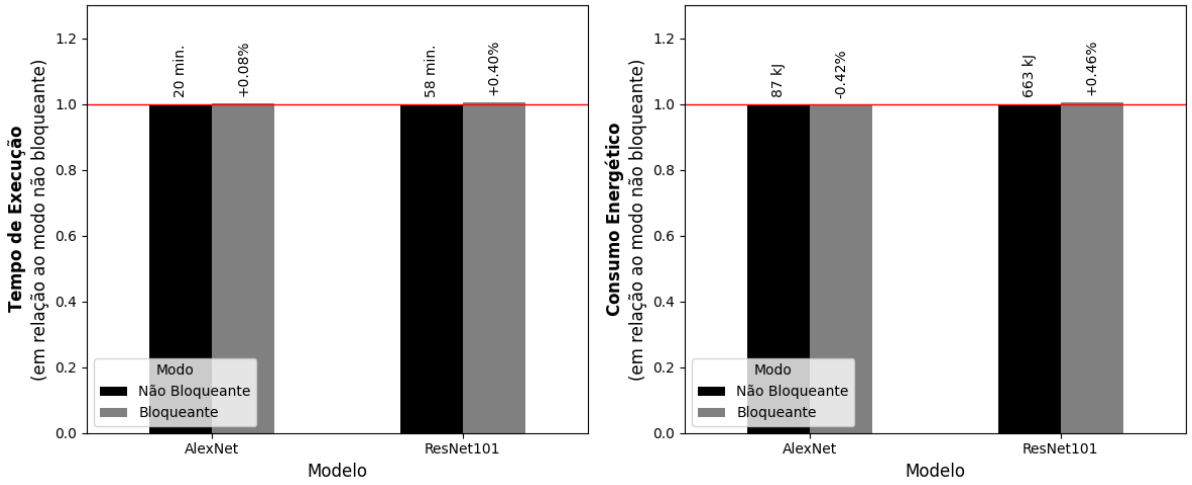


Figura 9: **Tempo de execução e consumo energético da GPU associados ao treino de cada modelo de acordo com o modo de cópia.** Experiências no ambiente de teste *Commodity* armazenando o *dataset* localmente.

Na figura 9, são representados os valores de consumo energético e tempo de execução, para cada modelo, normalizados em relação aos valores obtidos com modo de cópia não bloqueante. De acordo com esta representação, podemos verificar que, ao alterar o modo de cópia para bloqueante, as variações registadas nas métricas mencionadas são negligenciáveis (≈0%).

Deucalion - Armazenamento Local. Do mesmo modo, no *Deucalion*, com o *dataset* armazenado localmente, os resultados com os diferentes modos de cópia também se mantiveram semelhantes para ambos os modelos tanto para treino numa única **GPU** (Figura 10) como para treino multi-**GPU** (Figura 11).

Utilizando o modo não bloqueante e treinando com uma única **GPU**, o treino do modelo AlexNet demorou 1 hora e consumiu 383 kJ, já o treino do modelo ResNet101 terminou após 1 hora e 39 minutos, consumindo um total de 2 202 kJ. Adicionalmente, verificamos que a mudança no modo de cópia para bloqueante levou a variações inferiores a 1% nas referidas métricas (Figura 10)

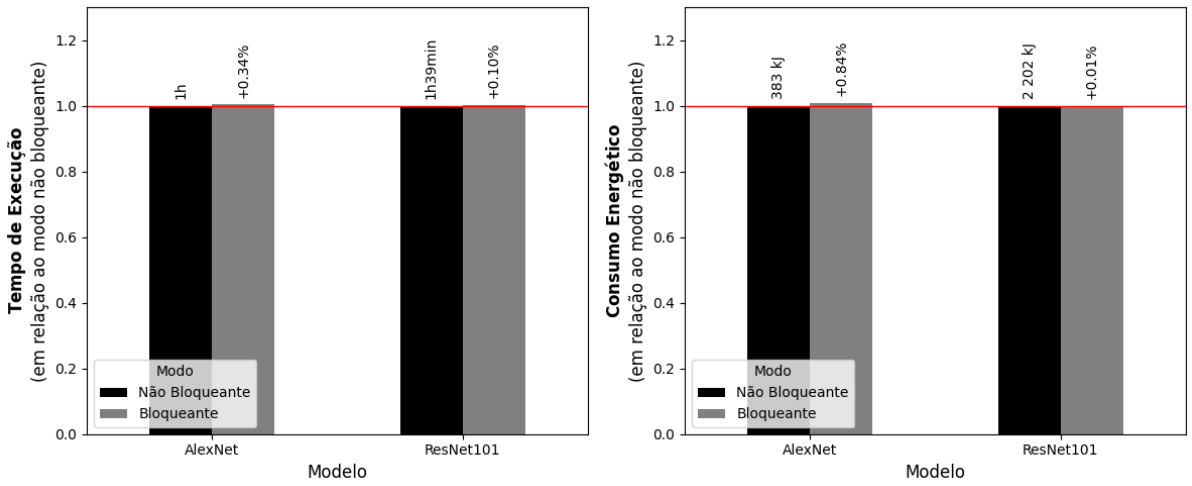


Figura 10: **Tempo de execução e consumo energético da GPU associados ao treino de cada modelo de acordo com o modo de cópia.** Experiências no ambiente de teste *Deucalion* recorrendo apenas a uma **GPU** para o treino e armazenando o *dataset* localmente.

Por outro lado, utilizando as quatro **GPU**s disponíveis, mas mantendo o modo de cópia não bloqueante, o treino do modelo AlexNet demorou 1 hora e 2 minutos com um consumo energético de 1 384 kJ, enquanto o treino do modelo ResNet101 durou 1 hora e foi registado um consumo de 3 042 kJ. Também as variações registadas neste cenário demonstram que não é necessariamente melhor, em termos de tempo de execução e consumo de energia, utilizar o modo de cópia não bloqueante (Figura 11).

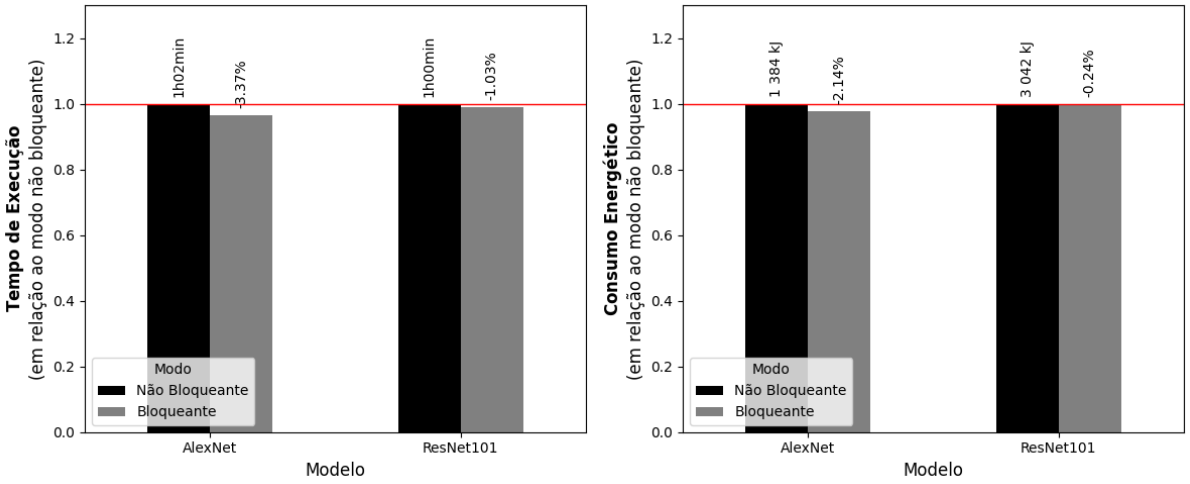


Figura 11: **Tempo de execução e consumo energético das GPUs associado ao treino de cada modelo de acordo com o modo de cópia.** Experiências no ambiente de teste *Deucalion* utilizando as 4 **GPU**s para o treino e armazenando o *dataset* localmente.

Deucalion - Armazenamento Remoto. Ainda no *Deucalion* mas com o *dataset* armazenado no dispositivo de armazenamento remoto (i.e.,sistema de ficheiros Lustre), foram registados resultados bastante semelhantes para os testes realizados com o modo de cópia bloqueante e não bloqueante. Neste caso, a variação registada é maior do que nos casos anteriores, no entanto, justifica-se pela variabilidade dos acessos ao armazenamento remoto. Isto é, como o armazenamento remoto é partilhado, pode ser acedido por outras cargas de trabalho e, dependendo do número de acessos em cada momento, a rede pode estar mais ou menos sobrecarregada, levando a variações no tempo de carregamento dos fragmentos do *dataset*. Assim, se uma das execuções acontecer durante um período de maior carga na rede, as leituras serão mais lentas e, consequentemente, o tempo de execução global será maior.

Neste cenário, usando o modo de cópia não bloqueante e utilizando apenas uma **GPU** para treinar, o treino do modelo AlexNet demorou 1 hora e 27 minutos consumindo 507 kJ, enquanto o modelo ResNet101 treinou durante 1 hora e 40 minutos com um consumo energético total de 2 220 kJ. Tal como representado na Figura 12, a mudança no modo de cópia para bloqueante não levou a variações significativas nem no tempo de execução nem no consumo energético.

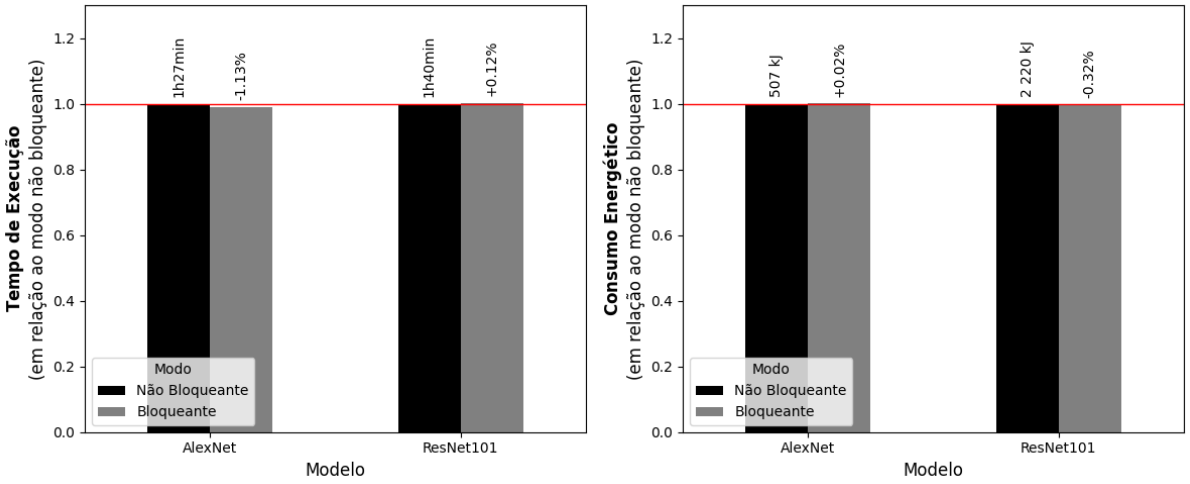


Figura 12: **Tempo de execução e consumo energético da GPU associados ao treino de cada modelo de acordo com o modo de cópia.** Experiências no ambiente de teste *Deucalion* recorrendo apenas a uma **GPU** para o treino e armazenando o *dataset* remotamente.

Como se pode observar na Figura 13, ao treinar os modelos recorrendo às quatro **GPU**s com o modo de cópia não bloqueante, o treino do modelo AlexNet demorou cerca de 2 horas e 6 minutos com um consumo energético de 2 653 kJ, por sua vez, o treino do modelo ResNet101 terminou após 2 horas consumindo 4 288 kJ. Neste cenário, principalmente para o modelo AlexNet, foram registadas variações mais expressivas, mas que se justificam pela variabilidade nos acessos ao armazenamento.

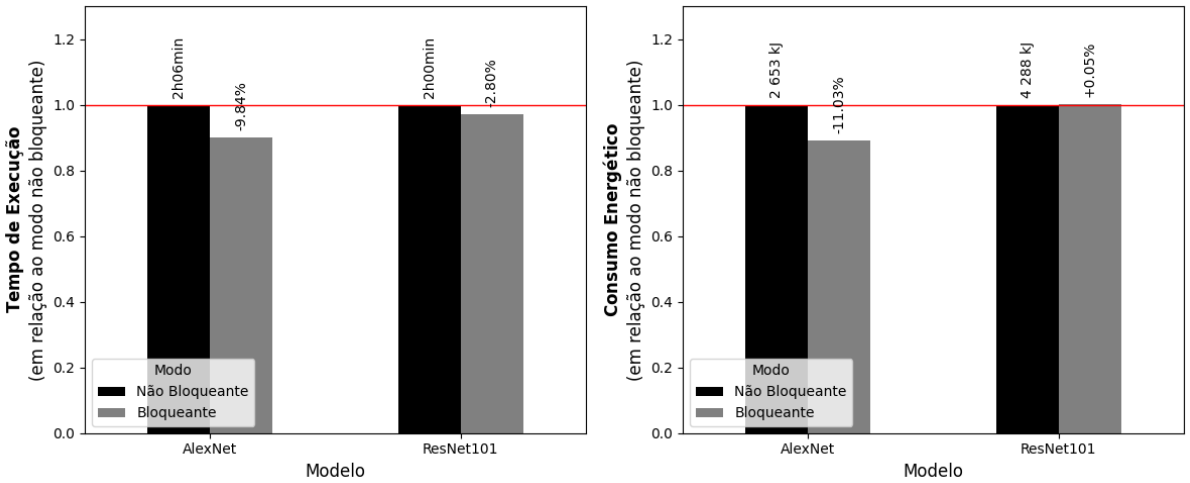


Figura 13: **Tempo de execução e consumo energético das GPUs associados ao treino de cada modelo de acordo com o modo de cópia.** Experiências no ambiente de teste *Deucalion* utilizando as 4 **GPU**s para o treino e armazenando o *dataset* remotamente.

Analisando a média do tempo de carregamento em cada uma das situações (Figura 14), podemos confirmar que a sua variação vai de encontro à variação do tempo de execução global. Por exemplo, o modelo AlexNet, quando treinado utilizando quatro **GPU**s, registou uma diminuição de 9.84% no tempo de execução quando o modo foi alterado para bloqueante (Figura 13) e podemos verificar que tal foi acompanhado por uma diminuição de 23.28% da média do tempo de carregamento, o que corresponde a uma diminuição de cerca de 56 milissegundos por iteração. Tal confirma que a variação registada no tempo de execução se deve à variação inserida pelo acesso ao armazenamento remoto.

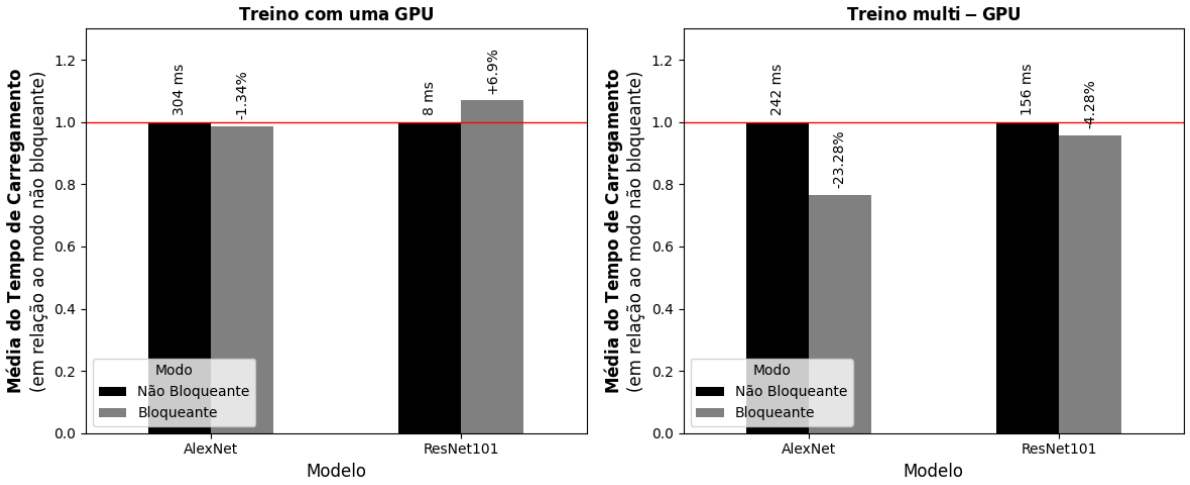


Figura 14: **Média do tempo de carregamento de acordo com o modelo, número de GPUs usadas e modo de cópia, com o dataset armazenado remotamente.** Experiências no ambiente de teste *Deucalion* armazenando o *dataset* remotamente.

Adicionalmente, o desvio padrão do tempo de execução global do treino dos modelos entre execuções do mesmo teste também corrobora esta teoria. Como podemos analisar na Tabela 4, o desvio padrão do tempo de execução quando é usado o armazenamento remoto pode ser bastante elevado, principalmente para o modelo AlexNet que, por ser intensivo em movimentação de dados, sofre de maior degradação do tempo de execução derivada dos acessos ao dispositivo remoto. Ainda, como verificamos anteriormente, também é este modelo que apresenta variações mais expressivas no tempo de execução e consumo energético quando utilizados os diferentes modos de cópia (Figura 13).

Tabela 4: **Média e desvio padrão do tempo de execução por modelo, modo de cópia e modo de treino.** Experiências no ambiente de teste *Deucalion* armazenando o *dataset* remotamente

Modelo	Modo de Cópia	1GPU		Multi-GPU	
		Média do Tempo de Execução	Desvio Padrão do Tempo de Execução	Média do Tempo de Execução	Desvio Padrão do Tempo de Execução
AlexNet	Não Bloqueante	5 241 s	1 442.62 s	7 537 s	698.14 s
	Bloqueante	5 182 s	1 002.52 s	6 795 s	184.64 s
ResNet101	Não Bloqueante	5 995 s	5.56 s	7 219 s	56.37 s
	Bloqueante	6 002 s	9.53 s	7 017 s	168.51 s

Sumário. Estes resultados permitiram concluir que, independentemente da máquina, do dispositivo de armazenamento e do modelo a treinar, o tempo de execução e o consumo energético não são afetados pelo modo da cópia de dados entre a memória da **CPU** e da **GPU**. No entanto, a utilização do modo de cópia bloqueante permite ter um controlo mais preciso sobre cada fase da iteração. Assim sendo, daqui em diante, todos os testes foram feitos utilizando o modo bloqueante.

De notar que esta conclusão é verdadeira para quando a instrução de cópia dos dados entre a memória da **CPU** e da **GPU** é seguida imediatamente por uma instrução em que os dados a serem copiados são utilizados. Por precisar dos dados para executar a instrução seguinte, mesmo utilizando o modo não bloqueante, a cópia acaba por, de qualquer das formas, bloquear a execução do treino antes do início da instrução seguinte.

3.3 Limite de Frequência

A aplicação de diferentes frequências à **GPU**s logicamente leva a diferentes valores de tempo de execução e consumo energético, no entanto, a relação entre estas métricas é complexa. Assim, este cenário de teste serve para analisar o comportamento destas métricas em diferentes modelos e sobre diferentes configurações de frequência de atuação da **GPU**.

Commodity. De acordo com os resultados apresentados no gráfico à esquerda na Figura 15 (relativo ao tempo de execução), percebemos que o desempenho do modelo ResNet101 tem uma sensibilidade maior à frequência da **GPU**, já que apresenta aumentos mais expressivos no tempo de execução com a

diminuição da frequência. Este resultado deve-se à natureza mais intensiva em computação deste modelo que garante que a **GPU** tem sempre dados para processar, não havendo períodos longos de inutilização deste recurso. Assim, ao diminuir a frequência de atuação da **GPU**, esta fica mais lenta levando a que as fases que dela dependem (cópia e carregamento) se prolonguem e, consequentemente, o tempo global de execução aumente (Figura 16).

Por outro lado, isto não se verifica tão acentuadamente para o modelo AlexNet, sendo um modelo mais intensivo a nível do movimento de dados, a **GPU** fica inutilizada durante períodos mais longos. Assim, ao diminuir a frequência da **GPU**, as fases de cópia e computação são estendidas, aumentando a sua sobreposição com a fase de carregamento e, consequentemente, diminuindo os períodos de espera pelos dados, o que torna o desempenho deste modelo mais tolerante a frequências baixas (Figura 17).

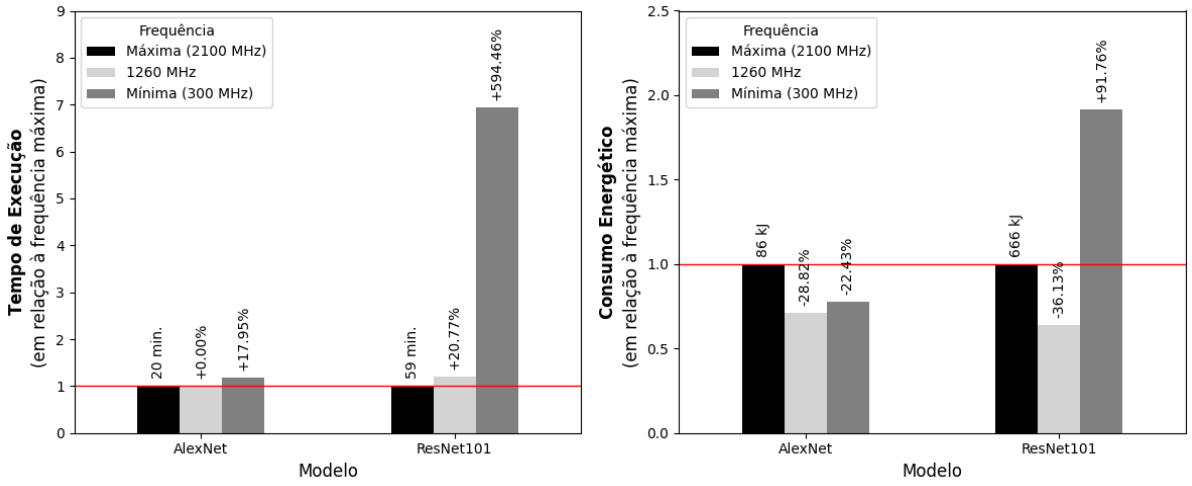


Figura 15: **Tempo de execução e consumo energético da GPU associados ao treino de cada modelo de acordo com diferentes configurações de frequência.** Experiências no ambiente de teste *Commodity* armazenando o *dataset* localmente.

Quanto ao consumo energético sabemos que, à medida que a frequência diminui, o consumo energético por unidade de tempo também diminui. Assim, para ambos os modelos, observamos uma diminuição do consumo energético na transição da frequência máxima para a intermédia (1260 MHz). No entanto, na transição da frequência intermédia para a mínima registamos um aumento do consumo energético, o que se justifica pelo aumento acentuado do tempo de execução entre estes dois cenários (Figura 15).

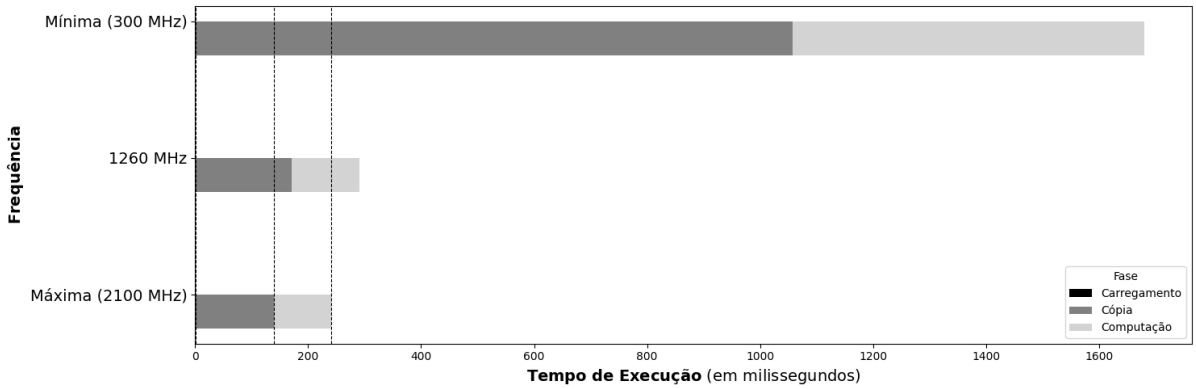


Figura 16: **Duração média das fases da iteração para o modelo ResNet101 sobre as diferentes configurações de frequência.** Experiência no ambiente de teste *Commodity* armazenando o *dataset* localmente.

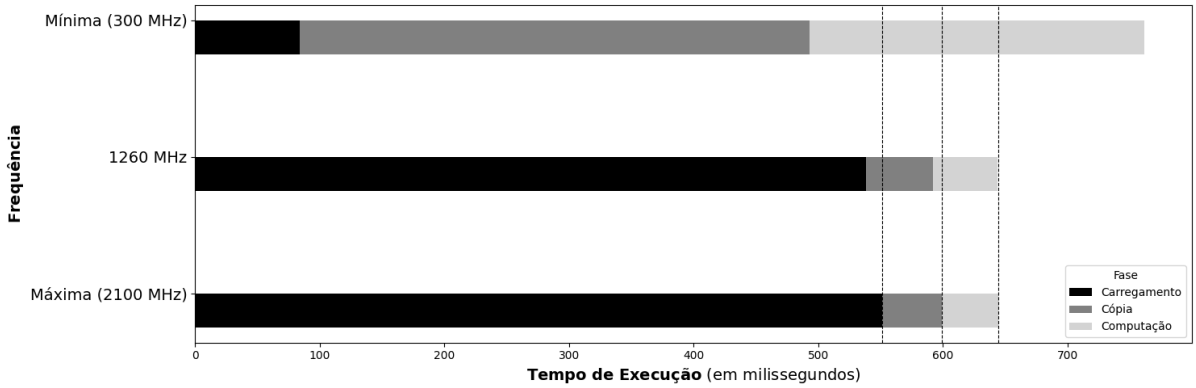


Figura 17: **Duração média das fases da iteração para o modelo AlexNet sobre as diferentes configurações de frequência.** Experiências no ambiente de teste *Commodity* armazenando o *dataset* localmente.

Sumário. Este cenário de teste comprova assim a relação complexa entre o tempo de execução, consumo energético global e frequência, já que, apesar de ser verdade que a frequência máxima garante o menor tempo de execução possível, não é verdade que outras configurações não consigam garantir o mesmo (Figura 15 - AlexNet 1260 MHz) nem é verdade que a frequência mínima garanta menor consumo energético do que as frequências mais altas.

Assim, para além de confirmarmos a relação complexa entre estas métricas, verificamos que esta relação depende ainda das características do modelo em questão. Deste modo, concluímos que não existe uma frequência única a aplicar durante o treino de todos os modelos.

3.4 Limite de Frequência por Fase

Tendo em conta que cada uma das três fases de uma iteração (carregamento, cópia e computação) implica a utilização de diferentes recursos a níveis diferentes, este cenário de teste tem como objetivo analisar a hipótese de que, com a aplicação de diferentes frequências em cada fase, é possível alcançar uma melhor eficiência energética do que aplicando uma frequência única, como no cenário anterior.

Fase de carregamento

Sendo a fase de carregamento uma fase na qual a **GPU** não tem qualquer tipo de participação, utilizar a frequência mínima da **GPU** nessa fase não deveria prejudicar o tempo de execução global do treino. Ainda, essa configuração deveria garantir um menor consumo energético, pelo que foi o primeiro cenário testado.

Commodity. Como podemos observar pela Figura 18, o treino do AlexNet durou 20 minutos incorrendo num consumo de 86 kJ, enquanto o treino do ResNet101 demorou 59 minutos e consumiu um total de 666 kJ. É possível ainda verificar que, ao aplicar a frequência mínima durante a fase de carregamento, no treino do modelo AlexNet, o consumo energético diminui significativamente (cerca de 31%) sem qualquer degradação do tempo de execução. No entanto, aplicando a mesma técnica durante o treino do modelo ResNet101, observamos um aumento do tempo de execução de 8% com uma poupança energética bem menor (de 4%).

Assim, podemos concluir que a teoria previamente apresentada se verificou para o modelo AlexNet mas não para o ResNet101. Isto deve-se ao facto da espera de dados no início de cada iteração para o último modelo ser, em média, menor que 1 milissegundo, como observado anteriormente (Figura 16). Ainda, o aumento da frequência não é imediato, sendo maior quanto maior é a diferença entre a frequência inicial e a final [32]. No exemplo apresentado, a intenção é que haja uma transição da frequência mínima para a máxima no final da primeira fase (de carregamento) e, por isso, devido ao facto anteriormente apresentado, durante os primeiros momentos da segunda fase (de cópia de dados entre as memórias da **CPU** e **GPU**) a frequência de atuação não é a máxima, o que prejudica o tempo global de execução. Para além disso, no caso do modelo ResNet101, também não se verifica uma poupança energética expressiva, isto, mais uma vez, devido à curta duração do período de espera por dados, que não é longa o suficiente para que a aplicação da frequência mínima leve a poupanças significativas de energia.

No entanto, podemos observar que esta estratégia funcionou para o modelo AlexNet, levando a uma

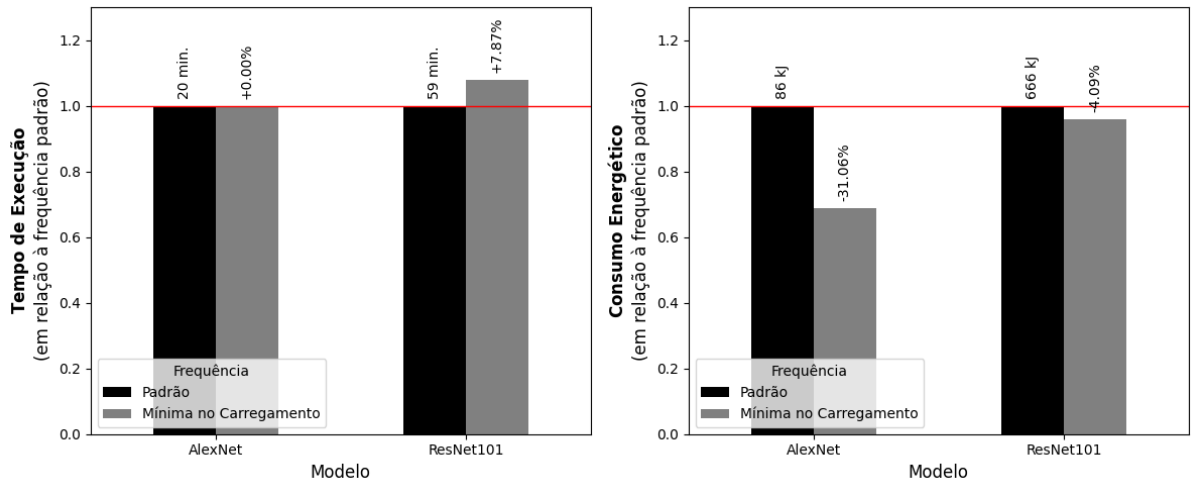


Figura 18: **Tempo de execução e consumo energético da GPU associados ao treino de cada modelo com a frequência padrão e com a frequência mínima durante o carregamento.** Experiências no ambiente de teste *Commodity* armazenando o *dataset* localmente.

diminuição expressiva do consumo energético (de 31.06%) sem prejuízo para o tempo de execução. Contrariamente ao modelo anteriormente analisado, este apresenta um tempo de espera por dados longo (fase de carregamento na Figura 19), em média acima dos 500 milissegundos por iteração, o que, para a máquina em questão, já é um período longo o suficiente para levar a poupanças energéticas mais acentuadas. Ainda, neste caso, a segunda fase (cópia) é também afetada pelo facto da mudança de frequência não ser imediata, no entanto, isto apenas leva à extensão dessa fase que, consequentemente, aumenta a sobreposição do final de uma iteração com a primeira fase da iteração seguinte, diminuindo o tempo de espera de dados (Figura 19).

Figura 19: **Duração média das fases de uma iteração para o modelo AlexNet.** Experiências no ambiente de teste *Commodity* armazenando o *dataset* localmente.

Assim, concluímos que esta estratégia funciona nos modelos em que o tempo de espera por dados é longo o suficiente para a frequência mínima levar a diminuições de consumo energético. Ainda, a duração deste período deve ser suficiente para garantir que o aumento da duração da fase seguinte, causado pelo facto da transição de frequências não ser imediata, causa maior sobreposição da computação com o carregamento do fragmento seguinte, diminuindo o tempo de espera por dados (fase de carregamento na Figura 19) e não aumentando o tempo de iteração. De notar que o período mínimo para garantir tais requisitos será dependente da máquina utilizada, já que o período de transição entre frequências é dependente do *hardware* e da diferença entre as frequências e estas, por sua vez, dependem também do *hardware*.

Fases de Cópia e Carregamento

Para avaliar o impacto que a aplicação de diferentes frequências tem nas diferentes fases da iteração, foi aplicada uma frequência diferente a cada 100 iterações do treino dos modelos e analisada a duração de cada uma das fases sobre cada configuração.

Commodity. Para o AlexNet, observamos que, com diminuição da frequência, dá-se também a diminuição do tempo de espera por dados (representada como fase de carregamento na Figura 20) enquanto que as restantes fases seguem a tendência contrária. Como foi explicado na secção anterior (Secção 3.3), tal deve-se ao facto da duração das fases de cópia e computação, por serem dependentes da **GPU**, aumentar e, consequentemente, a sobreposição do final de uma iteração com o carregamento do fragmento seguinte também aumentar, levando a uma diminuição do período de espera por dados pela parte da **GPU**.

Adicionalmente, analisando a Figura 20, verificamos que, com a diminuição da frequência, a duração da fase de cópia aumenta mais do que a duração da fase de carregamento e, por isso, podemos concluir que a frequência tem impactos diferentes nestas duas fases. Por exemplo, com a frequência de 510 MHz, há um aumento de 226% da duração da fase de cópia e um aumento de 36% da duração da fase de computação em relação às durações registadas com a frequência máxima (2100 MHz).

De notar que este gráfico demonstra ainda que nem sempre configurações de frequência mais baixas garantem menor consumo energético, mesmo que mantenham o tempo de iteração semelhante (na Figura 20, frequências 345 MHz e 870 MHz).

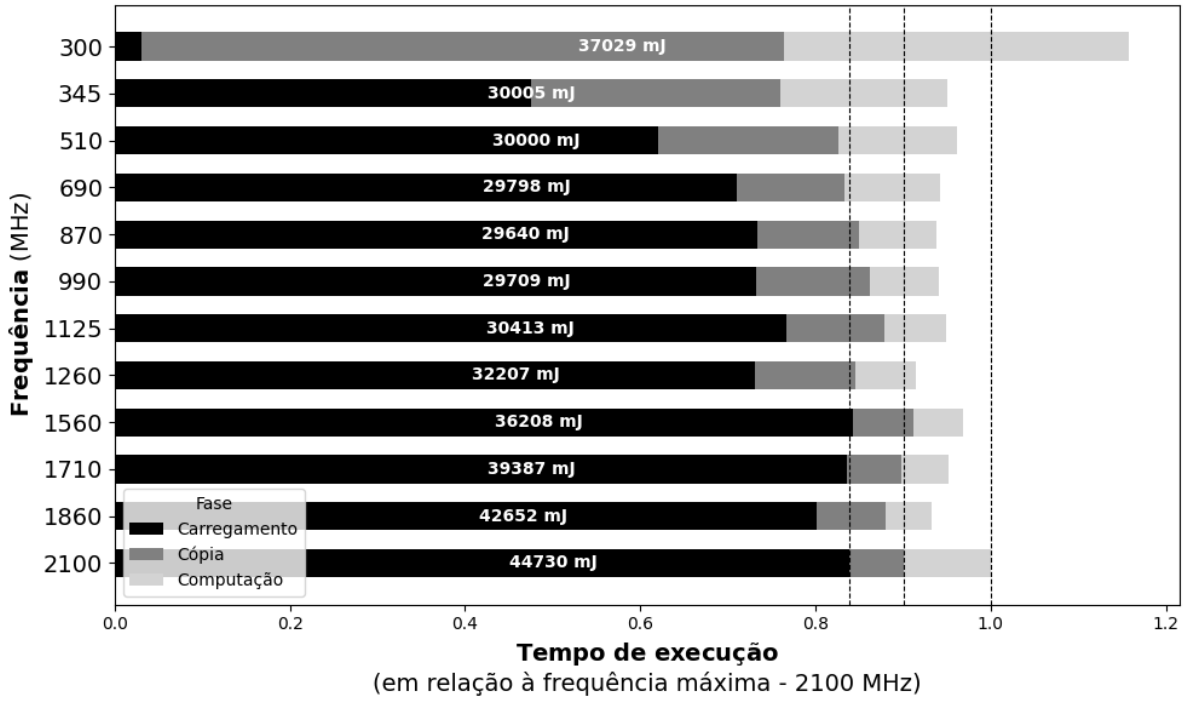


Figura 20: **Duração média das fases de uma iteração do AlexNet sobre diferentes frequências.** Experiência no ambiente de teste *Commodity* armazenando o *dataset* localmente.

Por sua vez, o ResNet18, que tal como o AlexNet, com um longo período de espera por dados, apresenta o mesmo comportamento, revelando o aumento da sobreposição de uma iteração com o carregamento do fragmento seguinte (representado pela diminuição da fase de carregamento na Figura 21) e um efeito mais expressivo da diminuição da frequência sobre a fase de cópia de dados do que sobre a fase de computação. Comparando, por exemplo, a duração das fases com a frequência máxima e com a frequência de 510 MHz, houve um aumento de 293% da duração fase de cópia e de 68% da duração da fase de computação.

Por fim, o mesmo teste foi feito durante o treino do modelo ResNet101 e, por este modelo ser mais intensivo em computação, contrariamente aos modelos anteriores, não se observa o aumento da sobreposição de uma iteração com o carregamento do fragmento seguinte, uma vez que a sobreposição já é quase total com a frequência máxima. No entanto, também neste caso, observamos que a diminuição da frequência tem maior impacto na fase de cópia (Figura 22). Analisando a duração das fases quando aplicada a frequência máxima e a frequência de 510 MHz, observamos um aumento de 179% na duração da fase de cópia e de 73% na duração da fase de computação.

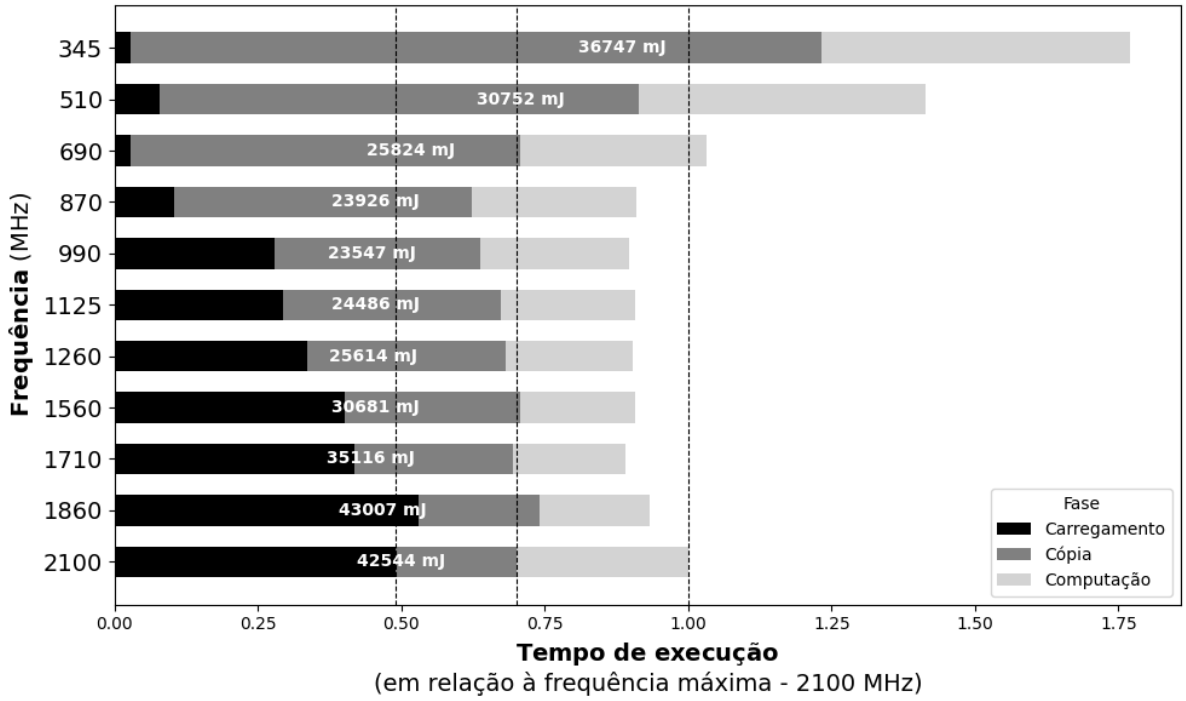


Figura 21: **Duração média das fases de uma iteração do ResNet18 sobre diferentes frequências.** Experiência no ambiente de teste *Commodity* armazenando o *dataset* localmente.

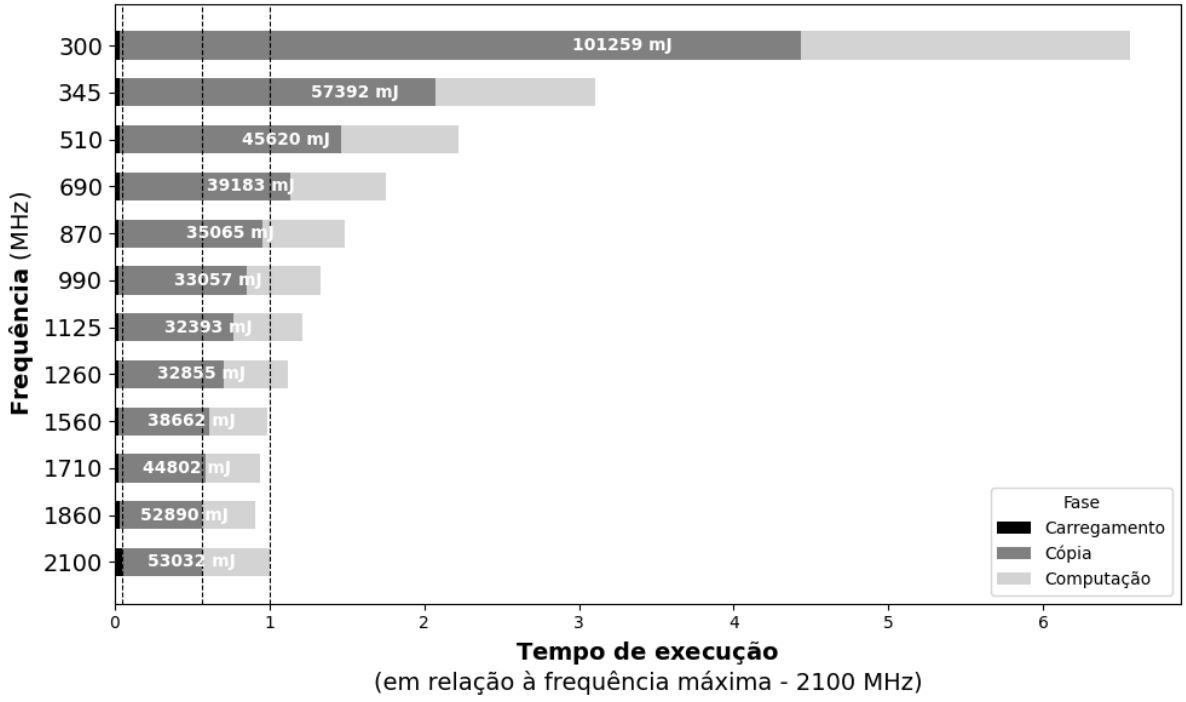


Figura 22: **Duração média das fases de uma iteração do ResNet101 sobre diferentes frequências.** Experiência no ambiente de teste *Commodity* armazenando o *dataset* localmente.

Sumário. Assim, a partir destes resultados, concluímos que a mesma frequência tem um impacto diferente sobre as diferentes fases de uma iteração, pelo que, não existe uma única frequência ótima para todas as fases.

3.5 Lições Aprendidas

Este estudo experimental resultou em três lições aprendidas que se revelaram pontos-chave para decisões a nível da arquitetura e funcionamento do sistema a desenvolver.

Lição 1. O primeiro cenário de teste permitiu-nos perceber que, de forma a garantirmos que controlamos o início e fim de cada fase da iteração, podemos utilizar cópias de dados bloqueantes sem que tal prejudique o desempenho do treino do modelo.

Lição 2. Por sua vez, o segundo cenário mostrou-nos que diferentes modelos sofrem diferentes efeitos com a mudança das configurações de frequência da **GPU**. Assim, as características da carga de trabalho devem ser tidas em conta aquando da escolha de uma frequência de atuação.

Lição 3. Já o último cenário de teste, mostrou-nos que uma iteração do treino do modelo apresenta diferentes fases que reagem de forma diferente às diferentes frequências. Estes resultados permitem-nos concluir que não existe uma única frequência ótima para todas as fases de uma iteração.

Capítulo 4

WAFT

Neste capítulo é apresentado o WAFT, *Workload-Aware Frequency Tunning*, um novo sistema que permite diminuir o consumo energético das **GPU**s associado ao treino de modelos sem comprometer o desempenho do mesmo. O WAFT ajusta dinamicamente a frequência da **GPU** tendo em conta as diferentes fases de uma iteração. Este ajuste permite otimizar o consumo energético sem prejudicar o tempo de execução, adaptando-se automaticamente às características do modelo e do *hardware*.

Com base no conhecimento obtido através do estudo anteriormente apresentado, foram estabelecidos quatro princípios de desenho que levaram à arquitetura final do sistema que será explicada em detalhe neste capítulo.

4.1 Princípios de Desenho

O WAFT foi desenvolvido sobre quatro princípios de desenho: controlo energético adaptável, controlo de grão fino, extensibilidade e portabilidade.

Controlo energético adaptável. O sistema ajusta a frequência ao longo do tempo, respondendo assim a variações de desempenho, disponibilidade e utilização dos recursos utilizados. O desempenho do treino do modelo é continuamente analisado, pelo que alterações no ambiente de teste que afetem o desempenho do treino serão detetadas (p.ex., volatilidade do sistema de armazenamento). Ao detetar tal alteração, o WAFT atua de forma a adaptar o seu comportamento ao novo cenário.

Controlo energético por fases de treino. O sistema reconhece os momentos de início e fim das diferentes fases de uma iteração, aplicando diferentes frequências em cada uma delas de acordo com o algoritmo. Esta abordagem permite que a frequência em cada fase seja adequada à intensidade da carga que a mesma impõe sobre a **GPU**. Como consequência, o WAFT consegue identificar e minimizar os períodos de inutilização sem aumentar o tempo de execução global do treino. Adicionalmente, estes períodos, sendo inevitáveis, são reconhecidos como oportunidades para poupança energética.

Extensibilidade. O sistema pode facilmente evoluir através da adição de algoritmos diferentes. Foram desenvolvidos algoritmos para a escolha da configuração das frequências de atuação, no entanto, o WAFT permite o desenvolvimento e integração de novos algoritmos, assegurando a sua adaptabilidade a diferentes contextos e avanços tecnológicos.

Portabilidade. O sistema permite adicionar suporte para **GPU**s de diferentes fornecedores. As ferramentas de controlo e monitorização energética variam de acordo com o fornecedor da **GPU**, assim, o WAFT permite a integração das mesmas de forma a garantir compatibilidade com qualquer ambiente.

4.2 Arquitetura

O WAFT é composto por 3 principais componentes: o **Controlador**, o **Profiler** e o **Algoritmo de Controlo**. O Controlador é responsável por delegar tarefas e decidir qual algoritmo será aplicado. Por sua vez, o *Profiler* regista informações sobre o comportamento do modelo ao longo da primeira época para traçar o perfil do mesmo e suportar a tomada de decisão do Controlador. Por fim, o Algoritmo de Controlo é responsável por calcular as frequências a aplicar ao longo das restantes épocas de treino.

Para além disso, existem as componentes **Baseline**, **Alto Coeficiente de Variação (CV)** e **Baixo CV** que são implementações concretas de um Algoritmo de Controlo. Existem ainda as componentes auxiliares: **Aplicador de Frequência**, que aplica as frequências à **GPU** de acordo com a decisão do Algoritmo de Controlo, e o **Monitor de Energia** que, tal como o nome indica, recolhe métricas de consumo energético da **GPU**.

Por fim, foi desenvolvida a componente **Hardware Profiler** para garantir que o sistema terá um funcionamento adaptado ao modelo da **GPU** que será usada no treino dos modelos.

Assim, antes da primeira execução do WAFT num novo ambiente de teste, deve ser utilizado o *Hardware Profiler*, para configurar o sistema de acordo com o *hardware* utilizado (identificado com o número ⓿ na Figura 23). Após este passo, o sistema está pronto para ser utilizado para otimizar o consumo energético do treino de modelos de forma adaptada ao *hardware*. Para tal, o Controlador, durante a primeira época de treino, informa o *Profiler* sobre o progresso do treino (representado pelo número ❶ na Figura 23) e, por sua vez, o *Profiler* regista métricas sobre este progresso, recorrendo ao Monitor de Energia (identificado pelo número ❷ na Figura 23). Ao terminar a primeira época, o Controlador decide que Algoritmo de Controlo aplicar e passa a informar esta componente do progresso do treino (representado pelo número ❸ na Figura 23). O Algoritmo de Controlo regista essa informação assim como métricas de consumo energético, recorrendo ao Monitor de Energia (processo identificado com o número ❹ na

Figura 23 e, com base em toda a informação recolhida, aplica a frequência que considera mais adequada ao longo do tempo através do Aplicador de Frequência (passo representado pelo número ❺ na Figura 23).

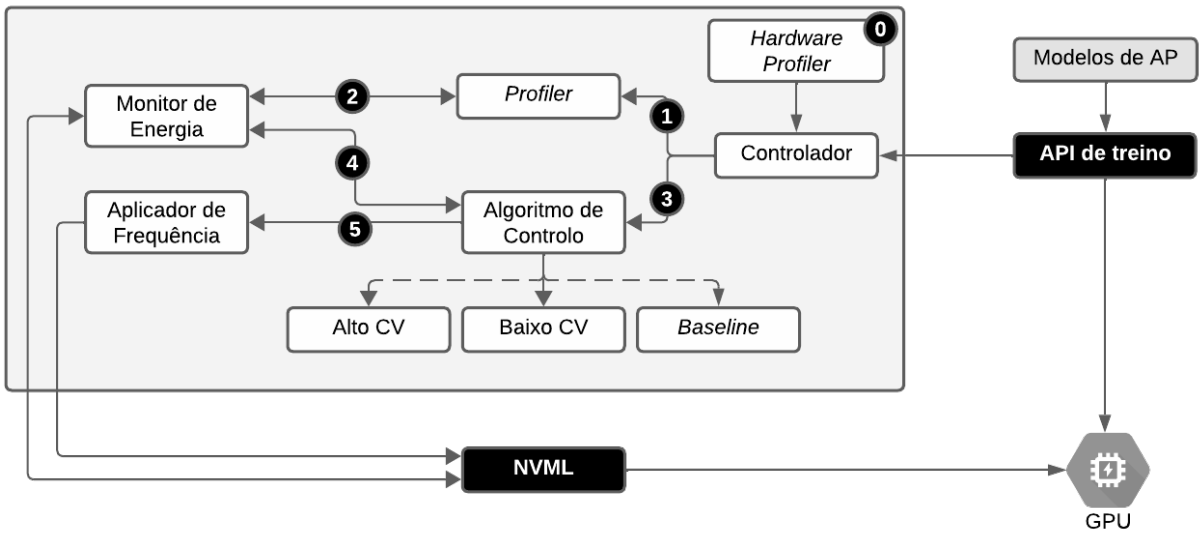


Figura 23: Arquitetura geral do sistema.

4.2.1 Hardware Profiler

O *hardware profiler* serve para adaptar a execução do WAFT ao modelo de **GPU** utilizado no treino, já que este avalia qual é o período mínimo para o qual é justificável fazer a alteração da frequência da **GPU** para o valor mínimo suportado. Isto é, qual é a duração mínima de um período de inutilização, entre períodos de alta utilização da **GPU**, para o qual se registam poupanças energéticas ao aplicar a frequência mínima suportada pela **GPU** enquanto está inativa.

O valor produzido por esta componente deve ser utilizado na criação do Controlador para adaptar o seu comportamento ao modelo da **GPU** usada. Este valor permite ao Controlador compreender se, durante o treino, o algoritmo deve ou não aplicar a frequência mínima com base na duração prevista de um período de inutilização.

Deste modo, a atuação desta componente é necessária apenas antes da primeira execução do sistema num novo ambiente de teste (identificado pelo número ❶ na Figura 23) pois produz um valor dependente apenas do *hardware* e que, por isso, se mantém entre execuções. Ainda, no caso de treino multi-**GPU**, em ambientes homogéneos (i.e., em que todas as **GPU**s são do mesmo modelo), basta utilizar o *hardware profiler* para uma delas, já que o valor obtido deverá ser o mesmo para **GPU**s iguais.

4.2.2 Controlador

Após a execução do *hardware profiler* pode dar-se início à primeira execução do sistema. Para tal, deve ser criada uma instância de Controlador por cada uma das **GPU**s que participe no treino do modelo, já que o Controlador foi desenvolvido para controlar uma só **GPU** (Figura 24).

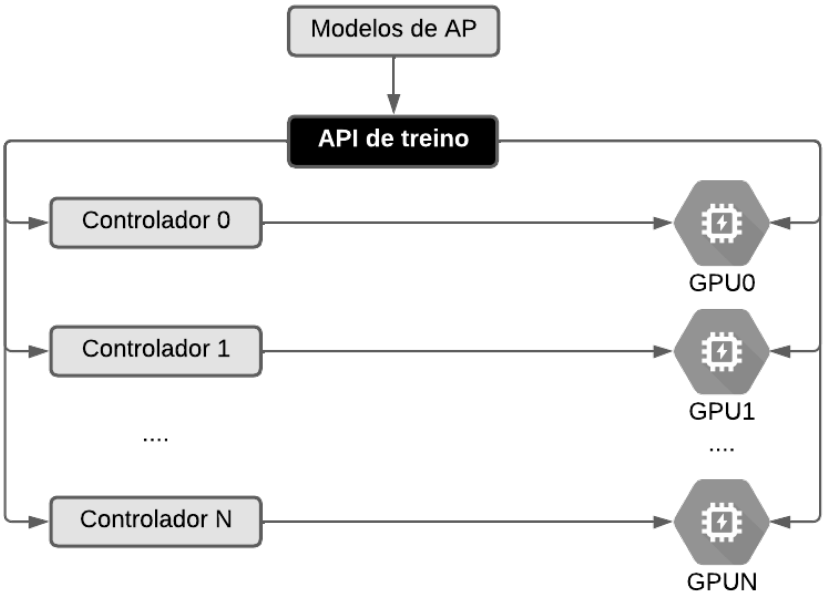


Figura 24: Aplicação do WAFT com várias **GPU**s envolvidas no treino.

Estas instâncias devem ser utilizadas para invocar as funções, disponibilizadas no Controlador, que devem assinalar no código do treino o início e fim de cada época e de cada fase dentro de uma iteração: carregamento, cópia e computação. No entanto, a única responsabilidade do Controlador nestas funções é delegá-las ao atual responsável: *Profiler* durante a primeira época (processo identificado pelo número ❶ na Figura 23) e Algoritmo de Controlo, durante o restante treino (representado pelo número ❸ na Figura 23).

Para além de delegar a tarefa de lidar com a informação sobre o progresso do treino, o Controlador deve também decidir que algoritmo deve ser utilizado para o modelo em questão. Assim, cria as estruturas para o armazenamento de métricas sobre a primeira época que são preenchidas pelo *Profiler*. Quando este termina a sua execução, no final da primeira época, o Controlador analisa as métricas recolhidas de forma a tomar a decisão sobre o Algoritmo de Controlo a aplicar. Após decidir, cria o Algoritmo de Controlo que considera mais adequado, dá início à sua execução e passa a delegar-lhe as tarefas.

Devido à existência do Controlador, é possível, através de mudanças localizadas, adicionar novos algoritmos de controlo e alterar as métricas consideradas na tomada de decisão sobre o algoritmo a utilizar.

4.2.3 Profiler

Ao receber a informação sobre o progresso do treino do modelo (através da chamada das funções anteriormente mencionadas por parte do Controlador), esta componente regista a duração de cada iteração e respetivas fases, assim como o consumo energético por iteração. Tendo terminado a primeira época, pode fazer cálculos (p.ex., média, desvio padrão, etc.) sobre os dados recolhidos, que podem ser utilizados pelo Controlador para escolher o algoritmo a utilizar. No caso de treino de modelos que incluem uma fase de validação após cada fase de treino, esta também é analisada, mas separadamente, já que estas duas fases apresentam intensidades de computação diferentes.

A existência desta componente permite que o sistema recolha informação suficiente sobre as características do treino do modelo em tempo de execução de forma a escolher o melhor algoritmo a aplicar, já que diferentes modelos reagem de forma diferente à alteração da frequência. Ainda, para considerar diferentes métricas na escolha do Algoritmo de Controlo a usar, será apenas necessário adicionar a monitorização das mesmas nesta componente e adicionar no Controlador a sua análise.

4.2.4 Algoritmo de Controlo

Após o final da primeira época, as informações sobre o progresso do treino passam a ser comunicadas ao Algoritmo de Controlo. Ao receber tais informações, tal como o *Profiler*, regista a duração das fases de uma iteração e o consumo energético por iteração e, com base nestas métricas, deve decidir as frequências a aplicar em cada momento. O algoritmo pode ser desenhado para escolher diferentes frequências para cada fase da iteração, aplicando a frequência escolhida para uma fase quando recebe a informação de que a mesma vai começar. Ainda, tal como a componente anteriormente descrita, esta também identifica a fase de validação, caso exista, e pode, inclusivamente, decidir aplicar diferentes frequências, tendo em conta estas diferentes fases de uma época.

Esta componente pode ter diferentes implementações, correspondentes a diferentes algoritmos, e o Controlador pode criar diferentes instâncias de acordo com os critérios definidos. Deste modo, podem ser adicionados facilmente novos algoritmos, sendo apenas necessário desenvolver o código correspondente ao algoritmo e acrescentar a criação de uma instância do mesmo no Controlador, criação que pode ser condicional, com base em métricas recolhidas pelo *Profiler*, por exemplo.

4.2.5 Monitor de Energia e Aplicador de Frequência

Tanto o *Profiler* como o Algoritmo de Controlo recolhem métricas de energia ao longo do treino do modelo, recorrendo para tal ao Monitor de Energia. Adicionalmente, o Algoritmo de Controlo dá indicações sobre as frequências a aplicar à **GPU** que são postas em prática através do Aplicador de Frequência.

Estas componentes foram criadas para isolar a comunicação com a ferramenta de controlo e monitorização energética do restante sistema, o que facilita a possibilidade de expansão do sistema para considerar **GPU**s de diferentes fornecedores. As ferramentas de controlo e monitorização energética utilizadas dependem de fornecedor para fornecedor, desta forma, para permitir o uso do WAFT sobre **GPU**s de diferentes fornecedores, é apenas necessário implementar o Monitor de Energia e o Aplicador de Frequência utilizando uma ferramenta de monitorização e controlo compatível com as **GPU**s que se pretende usar.

4.3 Implementação

O WAFT foi desenvolvido em Python e a sua implementação está dividida por 9 ficheiros para facilitar a compreensão, manutenção e evolução do código. Cada ficheiro corresponde a uma componente representada na Figura 23 da secção anterior, existindo ainda um ficheiro onde são definidas algumas classes para armazenar dados e facilitar a sua consulta, assim como a realização de cálculos sobre os mesmos. Esta secção apresenta e explica os detalhes de implementação de cada componente.

4.3.1 Controlador

Para usar o WAFT, deve ser criada uma instância do Controlador por cada **GPU** utilizada no treino do modelo (Figura 24) e, de seguida, deve ser lançada uma *thread* para executar a função `start` do Controlador. Posteriormente, esta instância deve ser utilizada para invocar as funções disponibilizadas pelo Controlador (descritas na Tabela 5) para delimitar as várias fases do treino e de cada iteração, como demonstrado no *Listing* 1.

Assim, no início de cada época, nomeadamente no início da fase de treino, deve ser utilizada a função `start_train()`. Também no início de cada fase de validação, caso esta exista, deve ser utilizada a função `start_validation()` para que o Controlador identifique a fase adequadamente.

Adicionalmente, no código respeitante aos ciclos que compõem o treino e validação do modelo, em que cada iteração do ciclo corresponde a uma iteração de treino/validação, devem ser usadas as funções

Listing 1 Exemplo do código de treino com invocação das funções do Controlador.

```
for i in range(0, epochs):
    train(i)
    validate(i)
control.end()

def train(epoch):
    control.start_train()
    for i, (images, target) in enumerate(train_loader):
        control.end_load()
        images = images.to(device, non_blocking=False)
        target = target.to(device, non_blocking=False)
        control.end_move()
        # aplicação das computações da rede
        ...
        control.end_iteration()
    control.end_train()
```

`end_load()` para assinalar o fim do carregamento do fragmento, `end_move()` para assinalar o fim da cópia do fragmento para a memória da **GPU** e ainda a `end_iteration()` que, para além de assinalar o fim de uma iteração, assinala também o fim da fase de computação e o início da fase de carregamento da iteração seguinte.

Quando termina a fase de treino deve ser usada a função `end_train()` e, quando termina a fase de validação, caso exista, deve ser usada a função `end_validation()`. Por fim, quando todas as épocas tiverem sido terminadas, deve ser invocada a função `end()`.

Estas funções, no Controlador, têm a única e exclusiva função de invocar a função com o mesmo nome no responsável atual: *Profiler* durante a primeira época e Algoritmo de Controlo durante o restante treino. Assim, estas duas componentes devem implementar as suas versões destes métodos, sendo que no *Profiler* devem ter uma função puramente de monitorização, enquanto no Algoritmo de Controlo podem também ser utilizadas para alteração de frequência.

Para além destas funções, o Controlador implementa ainda a função `start()` que é a função utilizada para lançar a *thread* que dá início ao funcionamento do sistema. Nesta função, o Controlador começa a execução do *Profiler* e, quando esta termina, após o final da primeira época, o Controlador decide que Algoritmo de Controlo criar e dá início à sua execução. Para tomar a decisão sobre o Algoritmo de Controlo, o Controlador analisa duas métricas calculadas pelo *Profiler*: a média do tempo de carregamento por iteração e o **Coeficiente de Variação** do tempo de iteração.

A média do tempo de carregamento por iteração é utilizada para decidir se deve ser aplicada a

Função	Descrição
`start_train()`	Assinala o início de uma de uma época, nomeadamente da fase de treino.
`start_validation()`	Assinala o início da fase de validação.
`end_load()`	Assinala o fim da fase carregamento.
`end_move()`	Assinala o fim da fase de cópia.
`end_iteration()`	Assinala o fim da iteração e da fase de computação.
`end_train()`	Assinala o fim da fase de treino.
`end_validation()`	Assinala o fim da fase de validação.
`end()`	Assinala o fim do treino.

Tabela 5: Funções do Controlador.

frequência mínima à **GPU** na fase de carregamento durante o restante treino do modelo. Esta decisão considera ainda o valor produzido pelo *hardware profiler* que, tal como referido anteriormente, representa a duração mínima de um período de inutilização da **GPU** entre períodos de alta utilização para o qual compensa alterar a frequência da **GPU** para o valor mínimo. Como a fase de carregamento é um período de inutilização da **GPU**, se a média da duração dessa fase for maior do que o valor produzido pelo *hardware profiler*, então o Algoritmo de Controlo criado receberá uma indicação para aplicar a frequência mínima à **GPU** durante essa fase da iteração. Deste modo, quando o algoritmo recebe esta indicação, a fase de carregamento é considerada uma fase independente mas para a qual a frequência ótima já foi encontrada. Por outro lado, quando tal não acontece, significa que a duração dessa fase é demasiado curta, na máquina em questão, para que se observem benefícios com a alteração da frequência da **GPU**, pelo que o algoritmo passa a considerá-la parte da fase de cópia. Assim, em todos os casos, os algoritmos precisam apenas de encontrar duas frequências: uma para a fase de cópia e outra para a fase de computação.

Por sua vez, o **Coeficiente de Variação** do tempo de iteração é o fator que efetivamente decide o Algoritmo de Controlo a criar: a um modelo que apresente um alto coeficiente de variação (igual ou superior a 50%) deve ser aplicado o algoritmo Alto **CV** enquanto que, se um modelo apresenta um **Coeficiente de Variação** mais baixo, é aplicado o algoritmo Baixo **CV**.

4.3.2 Algoritmo de Controlo

Os algoritmos desenvolvidos consistem em duas fases: a amostragem e a monitorização, sendo que, para estas fases, o algoritmo considera apenas um subconjunto das frequências suportadas pela **GPU**.

A amostragem é a fase responsável por encontrar, com base no subconjunto de frequências considerado, a configuração de frequências ótima que, tal como referido anteriormente, consiste em duas frequências: uma para a fase de cópia e outra para a fase de computação. Não é necessário procurar a frequência a aplicar durante a fase de carregamento, pois ou esta é longa o suficiente para que seja aplicada a frequência mínima (e tal leve a poupança de energia) ou é demasiado curta, não compensando a alteração da frequência e é, por isso, considerada parte da fase de cópia. No primeiro caso, a frequência mínima é aplicada durante a fase de carregamento em ambas as fases do algoritmo (amostragem e monitorização).

Por sua vez, a fase de monitorização consiste em monitorizar as métricas relativas ao treino do modelo de forma a verificar se a configuração encontrada na fase anterior continua a ser adequada ao cenário atual e agir em conformidade caso tal não se verifique.

Amostragem

Para encontrar a configuração ótima, a amostragem avalia o comportamento do modelo em questão sobre diferentes configurações de frequência com base no subconjunto a considerar. Assim, o algoritmo começa por analisar uma amostra de iterações do treino do modelo executadas utilizando a frequência máxima para as duas fases (cópia e computação) e regista a média do tempo de iteração e do consumo energético por iteração, métricas que servem como referência daqui em diante. De seguida, é necessário explorar as outras combinações possíveis de frequências para as duas fases. Como tal, o algoritmo começa por fixar a frequência máxima para a fase de computação (identificado pelo número ❶ na Figura 25) e é diminuída a frequência de cópia até chegar ao valor mínimo (processo representado pelos números ❷, ❸, ❹ e ❺ da Figura 25).

Quando tal sucede, a frequência de computação é decrementada e fixada no segundo valor mais alto de frequência do conjunto considerado e a frequência de cópia volta ao valor máximo, representado pelo número ❻ na Figura 25. Novamente, a frequência de cópia vai sendo diminuída (passos ❼, ❽, ❾ e ❿ da Figura 25) e o processo repete-se até que ambas as frequências assumam o valor mínimo, o que significa que todas as combinações possíveis para as frequências das duas fases foram exploradas tendo em conta o subconjunto de frequências considerado.

Para cada uma destas combinações é registada a média do tempo de execução e energia consumida por iteração utilizando amostras de um número fixo de iterações. Com base nessas métricas é construído um *top* 5 das melhores combinações de frequências. Para uma combinação ser candidata a entrar neste *top* não pode exceder em mais de 5% (valor que pode ser alterado de acordo com o objetivo do utilizador) a

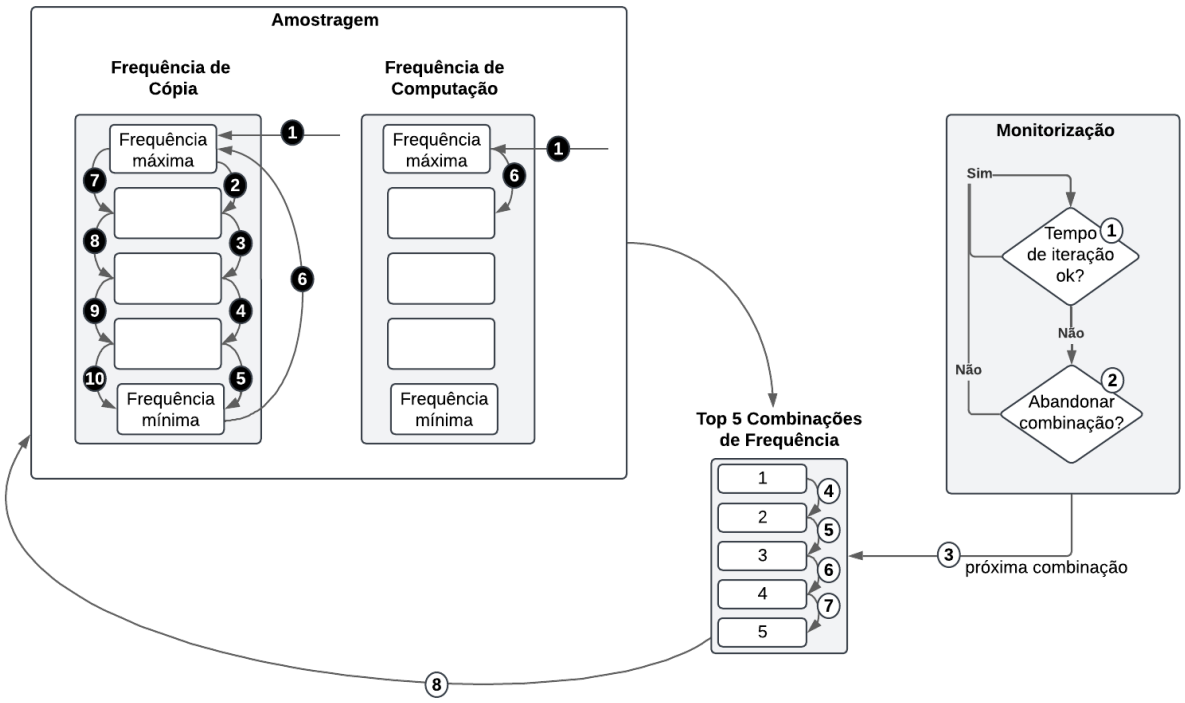


Figura 25: Fases dos algoritmos de controlo implementados.

referência recolhida para esta métrica (a média do tempo de iteração registada com a frequência máxima) e, dentro do *top*, as combinações são organizadas de acordo com a média de energia consumida por iteração, sendo a melhor combinação a que garante menor consumo energético.

Apesar de ser costumizável, foi definido, como padrão, o limite máximo de degradação do tempo de iteração de 5%, uma vez que se pretende garantir que a degradação máxima do tempo de execução global se fixe também nesse valor. Esta escolha deve-se ao facto de querermos mostrar que é possível obter bons resultados a nível de poupança de energia com uma baixa percentagem de aumento do tempo de execução, mas tendo em conta que a variação comum do tempo de execução global do treino de um modelo pode chegar aos 3% ou mais entre execuções [48].

No entanto, para certos modelos, existirão combinações que farão a média de tempo de iteração ultrapassar em muito mais do que 5% a referência recolhida pelo algoritmo. Assim, se, ao experimentar uma certa combinação, esta prejudicar em demasia o tempo de iteração, o algoritmo escolhe não explorar as combinações de frequências mais baixas que essa, pois também prejudicariam o tempo de iteração. Por exemplo, se tal acontecer no passo ❸ da Figura 25, não faz sentido proceder com os passos ❹ e ❺, por isso, o algoritmo passa diretamente para o passo ❻ da Figura 25. Os passos ❹ e ❺ consistiam no uso da mesma frequência de computação (máxima) mas em frequências mais baixas na fase de cópia (em comparação com o passo ❸) e, como tal, seria de esperar que, sendo a combinação do passo

❸ prejudicial ao tempo de iteração, também essas frequências fossem inadequadas para o modelo em questão. Se, por outro lado, a combinação do passo ❻ registar um tempo de iteração demasiado alto, então dá-se por terminada a fase de amostragem. Nesse caso, a frequência da fase de cópia é a máxima e, tanto esta como a frequência de computação, continuarão a diminuir se o processo de amostragem continuar, por isso, qualquer combinação explorada após esta, deverá levar a iterações mais longas e, como tal, não faz sentido continuar o processo.

O objetivo de ignorar algumas combinações de frequência durante fase de amostragem é diminuir o impacto desta fase no tempo de execução global, pois a parte do processo que é ignorada corresponderia a explorar combinações de frequências que podem prejudicar bastante o tempo de iteração e, consequentemente, aumentar o tempo de execução global. Ainda, ao prolongar o tempo de amostragem, diminuímos o período em que efetivamente estará aplicada a configuração ótima.

Quando se dá por finalizada a exploração das combinações de frequência possíveis é aplicada a combinação que se encontra em primeiro lugar no *top* construído.

Monitorização

Concluída a fase de amostragem, dá-se início à fase de monitorização que consiste na monitorização das métricas depois de aplicada a melhor combinação de frequências encontrada. Nesta fase, o objetivo é garantir que o tempo de iteração está em valores aceitáveis, isto é, abaixo do limite de degradação definido para o tempo de iteração em relação à referência recolhida na fase de amostragem. Assim, são analisadas as métricas relativas a amostras de número fixo de iterações e, se a média do tempo de iteração ultrapassar em mais de 5% o valor de referência registado na fase de amostragem (avaliação representada na Figura 25 com o número ①), o algoritmo deve averiguar se deve abandonar a combinação de frequências atual (decisão representada como número ② na Figura 25). Caso essa seja a decisão tomada pelo algoritmo, é aplicada a combinação na posição seguinte do *top* construído na fase de amostragem (passo ③ e ④ na Figura 25). De seguida, o algoritmo irá continuar a monitorizar o tempo de iteração (passos ① e ②) e, sempre que se justifique, o algoritmo continuará a mudar a configuração aplicada, experimentando as restantes combinações do *top* (passos ⑤, ⑥ e ⑦) até todas terem sido experimentadas. Quando tal sucede, é despoletada a fase de amostragem de novo, passo representado pelo número ⑧ na Figura 25.

A existência desta fase permite que o algoritmo seja adaptável a ambientes dinâmicos uma vez que, num cenário em que a disponibilidade de recursos muda, o tempo de iteração sofrerá alterações e, consequentemente, será abandonada a combinação utilizada até eventualmente ser despoletada a amostragem

que já terá em conta uma referência para o tempo de iteração condizente com o novo cenário.

Cada algoritmo utiliza esta estratégia (amostragem e monitorização) separadamente nas diferentes fases de uma época: treino e validação; considerando referências adequadas a cada fase e tentando encontrar a configuração ótima de frequências para cada uma delas pois, tendo intensidades de computação diferentes, terão também uma combinação de frequências ótima diferente. Assim, quando são invocadas as funções `end_train()` e `end_validation()`, são recuperados os valores de frequência a aplicar de acordo com a fase que se inicia, para que sejam utilizadas já na primeira iteração e o progresso do algoritmo relativo a essa fase seja retomado.

Alto Coeficiente de Variação

Para cada um dos algoritmos implementados foi necessário ter em atenção as características dos modelos aos quais são aplicados para a implementação das fases de amostragem e monitorização. O algoritmo Alto **CV** é aplicado em modelos cujo tempo de iteração apresenta um alto **Coeficiente de Variação** (igual ou superior a 50%, p.ex., como observado nos nossos testes em cenários de treino numa única **GPU**, classificamos os modelos AlexNet, ResNet18, e ShuffleNet como Alto **CV**).

Este algoritmo começa por definir as frequências a utilizar inicialmente, isto é, a combinação de frequências com as quais serão recolhidas as métricas de referência na fase de amostragem. Assim, é definida a frequência máxima para as fases de cópia e computação (linhas 7 e 8 do Algoritmo 1) e frequência mínima para a fase de carregamento (linha 3 do Algoritmo 1), caso essa tenha sido a indicação dada pelo Controlador (através da variável `min_load_freq` - linha 2 do Algoritmo 1), ou, a frequência máxima, caso contrário (linha 5 do Algoritmo 1), pois, neste caso, a frequência da fase de carregamento deverá ser a mesma aplicada durante a fase de cópia.

Posteriormente, à medida que o treino progride, isto é, à medida que as iterações do treino terminam, as métricas sobre elas recolhidas (variável `it`) são adicionadas a dois conjuntos (linhas 20 e 21 do Algoritmo 1): `samples` e `tendency` que representam conjuntos limitados de iterações cujas métricas são analisadas ao longo do algoritmo para decidir o seu comportamento. Assim, quando uma nova fase da época inicia (treino ou validação), todos os elementos nestes conjuntos são retirados (linhas 16 a 19 do Algoritmo 1) para que o algoritmo considere apenas métricas relativas à fase atual. Ainda, para guardar e analisar em todos os momentos o progresso relativo a cada uma destas fases, o algoritmo utiliza as estruturas `train_data` e `val_data` para armazenar a informação relativa à fase de treino e à fase de validação, respetivamente.

Assim que uma iteração é concluída, o algoritmo verifica a que fase da época pertence (linhas 11

a 15 do Algoritmo 1) e verifica se ainda está a decorrer a amostragem de frequências para essa fase (`data.sampling` na linha 22 do Algoritmo 1). Se tal se verificar e tiverem sido recolhidas métricas relativas a 7 iterações (`samples.full()` na linha 22 do Algoritmo 1), o algoritmo analisa-as. Se estas métricas tiverem sido recolhidas sobre as frequências inicialmente escolhidas (frequência máxima - linha 24 do Algoritmo 1), então a média do tempo de iteração e a média do consumo energético por iteração registadas vão servir como métricas de referência para esta fase do treino (linha 25 e 26 do Algoritmo 1). Ainda, se a média do tempo de iteração relativo à amostra de iterações já terminadas (`avg_time`) não ultrapassar em 5% (customizável, ω no Algoritmo 1) a referência para esta métrica (`data.ref_time`) e a média do consumo energético por iteração (`avg_energy`) for menor ou igual ao valor de referência (`data.ref_energy`), então esta combinação é candidata a entrar no *top* 5 de melhores combinações a considerar para esta fase da época (linhas 28 a 30 do Algoritmo 1). Assim, é utilizada a função `data.topk.add` que organiza as frequências numa fila com 5 elementos, em que a primeira combinação irá corresponder àquela que garante menor consumo energético. Por outro lado, se a média do tempo de iteração das iterações na amostra (`samples`) ultrapassar em mais que 50% (γ no Algoritmo 1) a referência para esta métrica então o algoritmo considera que não compensa explorar combinações de frequências mais baixas do que a usada na execução destas iterações (linhas 31 e 32 do Algoritmo 1). O tempo de iteração do treino destes modelos é bastante variável, pelo que as amostras podem, por vezes, refletir valores bastante acima da média real, mesmo utilizando a frequência padrão. Assim, para estes casos, o algoritmo opta por saltar passos da fase de amostragem apenas quando a referência é excedida em 50%, para tentar que esta fase não seja abandonada cedo demais, deixando combinações de frequências mais baixas por explorar, sendo que poderiam ser adequadas e alcançar maiores poupanças a nível energético. Por outro lado, quando a média do tempo de iteração não ultrapassa esse limite, o algoritmo deve prosseguir para a exploração da próxima combinação de frequências (linha 34 do Algoritmo 1). No entanto, se já todas as combinações tiverem sido exploradas, a amostragem dá-se por terminada e deve ser aplicada a melhor combinação de frequências encontrada (linha 38 do Algoritmo 1). Após concluir a análise das métricas relativas à iterações, estas devem ser retiradas do conjunto de amostras (linha 36 do Algoritmo 1) para que o algoritmo volte a analisar as métricas apenas quando tiverem sido concluídas 7 iterações com a nova combinação de frequências aplicada.

Por outro lado, se, ao analisar o progresso relativo à fase da época, verificar que a amostragem já está concluída, irá proceder à monitorização do desempenho do treino do modelo assim que tiverem sido recolhidas métricas sobre 7 iterações (linha 40 no Algoritmo 1). Quando tal sucede, verifica se a média de tempo de iteração da amostra respeita o limite imposto para a degradação do mesmo em relação

à referência para essa métrica (linha 42 do Algoritmo 1). Se o limite não for respeitado, o algoritmo começa por averiguar se a frequência aplicada durante a execução das iterações era a máxima (linha 44 do Algoritmo 1). Se tal se verificar, significa que a referência recolhida na fase de amostragem não está de acordo com as condições atuais e, por isso, é despoletada a amostragem para que sejam recolhidas novas referências (linha 45 do Algoritmo 1). No entanto, se estava a ser utilizada qualquer outra combinação de frequências, o algoritmo verifica a média do tempo de iteração registada nas últimas 30 iterações. Isto porque, analisando um número menor de iterações (`samples` no Algoritmo 1), o tempo de iteração pode ultrapassar o limite de degradação do tempo de execução, no entanto, isto não significar que a combinação utilizada não é adequada e ser apenas resultado da grande variação do tempo de iteração destes modelos. Por outro lado, analisando mais iterações (`tendency`), se a frequência aplicada for adequada, a média de tempo de iteração registada será aproximada do valor de referência. Mas, se também esta média ultrapassar em mais de 5% a referência (linha 46 do Algoritmo 1) então o algoritmo considera que a frequência aplicada deve ser alterada e procura a próxima combinação no *top* 5 (`data.topk.try_next`). Caso já tenha percorrido todas as frequências do *top*, é despoletada a amostragem para procurar novamente a frequência ótima (linhas 47 e 48 do Algoritmo 1). Ao terminar a análise das 7 iterações, o algoritmo retira as respetivas métricas do conjunto `samples` (linha 52 do Algoritmo 1), para não analisar novamente métricas relativas às mesmas iterações.

A utilização das métricas relativas às últimas 30 iterações (`tendency`), garante que o algoritmo está adaptado e preparado para o facto de o tempo de iteração ser muito variável mesmo quando a frequência utilizada é adequada. Ainda assim, o algoritmo continua capaz de compreender quando a frequência deve ser alterada. Pois, quando a frequência utilizada provoca um aumento do tempo de iteração indesejado, esse aumento vai ser refletido não só na média do tempo de iteração da amostra de iterações de menor tamanho (`samples`) como também na amostra de maior tamanho (`tendency`), pelo que o algoritmo irá detetar estes aumentos e abandonar a combinação de frequências atual.

A aplicação deste algoritmo a um modelo com baixo **Coeficiente de Variação** do tempo de iteração irá levar a que a amostragem seja mais longa do que o necessário. Nesses modelos, se uma combinação de frequências regista uma degradação do tempo de execução de 20%, é o suficiente para sabermos que nenhuma outra combinação que consista em frequências mais baixas irá respeitar o limite de degradação de 5%. Isto porque estes modelos apresentam um tempo de iteração bastante estável ao longo do tempo, pelo que um aumento nesta métrica só pode ser causado pela aplicação de uma frequência inadequada ao cenário atual. Assim, nestes modelos podemos e devemos ignorar combinações de frequência durante a fase de amostragem ao sinal de degradações do tempo de execuções menores do que no caso de

modelos com alto **CV**.

Ainda, nos modelos que apresentam baixo **CV**, a primeira iteração de treino é um *outlier*, pelo que considerá-la para a referência de tempo de iteração inflaciona esta métrica e, consequentemente, o algoritmo poderia causar uma degradação do tempo global de execução maior do que o que se esperava. Isto porque iria permitir que o tempo de iteração fosse até 5% maior do que a referência recolhida sendo que esta é uma versão inflacionada da média real do tempo de iteração do modelo.

Algorithm 1 Algoritmo para modelos com alto coeficiente de variação de tempo de iteração.

Inicialização: $samples[7]$; $tendency[30]$; $queue[1]$; $freqs = Frequencies()$; $\omega = 1.05$; $\gamma = 1.50$; $train\_data = Data()$; $val\_data = Data()$;

1: $queue \leftarrow queue.init()$
2: **if** min_load_freq **then**
3: $\quad freqs.load\_freq = get\_min\_supported\_freq()$
4: **else**
5: $\quad freqs.load\_freq = get\_max\_supported\_freq()$
6: **end if**
7: $freqs.copy\_freq = get\_max\_supported\_freq()$
8: $freqs.compute\_freq = get\_max\_supported\_freq()$
9: **while** $True$ **do**
10: $\quad it \leftarrow queue.dequeue()$
11: $\quad$ **if** $it.is\_validation()$ **then**
12: $\quad\quad data \leftarrow val\_data$
13: $\quad$ **else**
14: $\quad\quad data \leftarrow train\_data$
15: $\quad$ **end if**
16: $\quad$ **if** $it.it\_no = 1$ **then**
17: $\quad\quad samples.clear()$
18: $\quad\quad tendency.clear()$
19: $\quad$ **end if**
20: $\quad samples.push(it)$
21: $\quad tendency.push(it)$

```
22:     if data.sampling & samples.full() then
23:         avg_time, avg_energy ← samples.averages()
24:         if freqs.freq_max() then
25:             data.ref_time = avg_time
26:             data.ref_energy = avg_energy
27:         end if
28:         if avg_time < ω * data.ref_time & avg_energy ≤ data.ref_energy then
29:             data.topk.add(Frequencies(freqs, avg_time, avg_energy)
30:         end if
31:         if avg_time > γ * data_ref_it_time then
32:             data.skip_freq()
33:         else
34:             freqs ← data.find_next_sampling_freqs()
35:         end if
36:         samples.clear()
37:         if not data.sampling then
38:             data.apply_best_combo()
39:         end if
40:     else if samples.full() then
41:         avg_time, avg_energy ← samples.averages()
42:         if avg_time > ω * data.ref_time then
43:             tendency_time, _ = tendency.averages()
44:             if freqs.freq_max() then
45:                 data.reset()
46:             else if tendency.full() & tendency_time ≥ ω * data.ref_it_time then
47:                 if not data.topk.try_next() then
48:                     data.reset()
49:                 end if
50:             end if
51:         end if
52:         samples.clear()
53:     end if
54: end while
```

Baixo Coeficiente de Variação

Este algoritmo, Baixo **CV**, foi desenhado para modelos cujo tempo de iteração é mais estável ao longo de todo o treino, ou seja, observa-se um **Coeficiente de Variação** menor (inferior a 50%), como os modelos ResNet101 e VGG19 que são classificados como Baixo **CV** nos nossos testes em cenários de treino em uma única **GPU**.

Tal como no algoritmo anterior, este começa também por definir as frequências a utilizar inicialmente, que serão utilizadas para a recolha das métricas de referência na fase de amostragem. Desta forma, define que a frequência a aplicar durante as fases de cópia e computação é a máxima suportada (linhas 7 e 8 do Algoritmo 2) e, caso o Controlador assim o defina (através da variável `min_load_freq`), atribui a frequência mínima à frequência a usar durante a fase de carregamento (linha 3 do Algoritmo 2), caso contrário deve ser usada a frequência máxima (linha 5 do Algoritmo 1), pois, nesse caso, a frequência da fase de carregamento será a mesma da fase de cópia.

Com o avanço do treino do modelo, o algoritmo vai adicionando as métricas recolhidas sobre as iterações conforme estas terminam a um conjunto (linha 19 do Algoritmo 2) que representam amostras de iterações nas quais o algoritmo irá basear as suas decisões ao longo do tempo. Ainda, homologamente ao funcionamento do algoritmo anterior, este algoritmo utiliza estruturas para armazenar continuamente o progresso relativo a cada fase de uma época: treino e validação (`train_data` e `val_data`). Adicionalmente, quando uma destas fases se inicia são descartadas as amostras (`samples`) relativas às iterações da fase anterior e ainda à primeira iteração da fase atual (linha 16 a 18 do Algoritmo 2). Isto porque, apesar do tempo de iteração destes modelos ser bastante estável existe uma exceção: a primeira iteração de cada fase (de validação e treino). Nesta iteração é carregado o primeiro fragmento do *dataset* e, neste caso, não há sobreposição com a iteração anterior, pelo que a duração desta iteração é muito maior do que as restantes por ser fortemente afetada pelo tempo de carregamento. Assim, essas iterações não são analisadas pelo algoritmo pois, se fossem consideradas para os cálculos, iriam inflacionar os valores de referência durante a amostragem e, durante a monitorização, existia o risco do algoritmo abandonar a frequência previamente escolhida precipitadamente devido à média do tempo de iteração ter aumentado consideravelmente.

Quando algoritmo verifica que a amostragem para a fase da época atual ainda não terminou (`data.sampling` na linha 20 do Algoritmo 2), se já tiverem sido recolhidas métricas sobre 5 iterações, então o algoritmo analisa-as. Se essas iterações foram executadas aplicando as frequências inicialmente definidas (frequência máxima - linha 23 do Algoritmo 2) então, as métricas relativas ao tempo de execução e consumo energético destas iterações passam a ser considerados como referência pelo algoritmo (linhas

24 a 26 do Algoritmo 2). Ainda, se a média de tempo de iteração registada não ultrapassar em mais que 5% (ω) referência recolhida para esta métrica e a média de consumo energético por iteração também for inferior à respetiva referência, a combinação de frequências aplicadas durante a execução das 5 iterações é considerada candidata a pertencer ao *top* 5 (linhas 28 a 30 do Algoritmo 2). Assim, é utilizada a função `data.topk.add` que organiza as combinações candidatas tendo em conta as correspondentes métricas de energia e tempo por iteração, criando uma fila de 5 combinações em que a primeira é aquela que garante menor consumo energético.

No entanto, se a análise dessas iterações revelar que a média de tempo de iteração ultrapassa em mais de 20% (γ) a referência para o tempo de iteração, então o algoritmo decide que não deve explorar combinações de frequências mais baixas do que as aplicadas. Ao contrário dos modelos com alto **CV**, nestes, se o limite de degradação do tempo de iteração for ultrapassado, há uma grande probabilidade de que a causa seja o uso de uma combinação de frequências inadequada às condições atuais do ambiente. Assim, o critério para ignorar combinações de frequências mais baixas na fase de amostragem é mais agressivo do que para os modelos anteriores (linha 31 do Algoritmo 2). Esta é também a razão pela qual podem ser analisadas amostras de menor número de iterações pois, mesmo com uma amostra de menor tamanho, é possível obter uma média do tempo de iteração representativa dos efeitos da frequência no modelo, devido ao baixo **Coeficiente de Variação** desta métrica.

Ainda, se a referência para o tempo de iteração não for excedida em mais de 20%, é explorada a próxima combinação de frequências da amostragem (linha 34 do Algoritmo 2). No entanto, se já todas tiverem sido exploradas e, por isso, a amostragem se der por terminada, deve ser aplicada a melhor combinação encontrada (linha 37 a 39 do Algoritmo 2). Adicionalmente, após a análise das 5 iterações, as suas métricas devem ser retiradas do conjunto `samples` para que as próximas iterações que pertençam a este conjunto sejam apenas aquelas em que já foi aplicada a nova configuração de frequências (linha 36 do Algoritmo 2).

Por outro lado, quando o algoritmo percebe que a amostragem já está concluída procede à análise das métricas de tempo de iteração a cada 5 iterações executadas (linha 40 do Algoritmo 2). Começa por verificar se a média do tempo de iteração não ultrapassa em mais de 5% a referência para esta métrica (linha 43 do Algoritmo 2). Se tal se verificar e a frequência atualmente aplicada for a máxima, então é despoletada a amostragem, pois a referência do tempo de iteração que está a ser utilizada não representa o estado atual do sistema e por isso é necessário recolher uma nova referência e encontrar a frequência ótima dadas as novas condições (linhas 44 e 45 do Algoritmo 2). Quando isto sucede sobre qualquer outra frequência, há uma grande probabilidade dessa frequência não ser adequada ao cenário de treino atual,

seja por ter havido uma alteração na disponibilidade de recursos ou simplesmente porque a frequência foi indevidamente escolhida na fase de amostragem. No entanto, pode ter-se registado uma iteração com uma duração maior devido a um gargalo temporário na leitura do fragmento, por exemplo. Assim, para descartar essas situações, verificamos o desvio padrão. Se o desvio padrão do tempo de iteração da amostra for 5% maior que o valor de referência para essa métricas (valor registado na amostra de iterações executadas com a frequência máxima na fase de amostragem) (linha 46 do Algortimo 2) então consideramos que, o facto da média do tempo de iteração ter ultrapassado o limite de degradação, se deve a um *outlier* e, por isso, é mantida a frequência. O desvio padrão reflete a dispersão dos dados em relação à média, assim, o desvio padrão registado com a frequência máxima deve representar a dispersão típica do tempo de iteração para o modelo em questão, pelo que, se o valor de desvio padrão na amostra é maior do que o valor de referência então podemos considerar que existe pelo menos um *outlier* na amostra recolhida. Por outro lado, se a frequência for de facto inadequada, a média do tempo de iteração irá aumentar mas o desvio padrão deve manter-se, já que a frequência deverá aumentar o tempo de iteração para todas as iterações da amostra. Pelo que, se o desvio padrão revela uma dispersão menor ou similar à registada com a frequência máxima (linha 46 do Algoritmo 2) significa que o aumento na média do tempo de iteração reflete o uso de uma frequência inadequada e não apenas um *outlier*. Assim, o algoritmo irá aplicar a próxima frequência no *top*, no entanto, se este já tiver sido totalmente percorrido, então é despoletada a amostragem pois, as frequências escolhidas anteriormente nessa fase, mostraram-se inadequadas durante a monitorização (linhas 47 a 49 do Algoritmo 2).

Se este algoritmo fosse aplicado a modelos cujo **Coeficiente de Variação** do tempo de iteração é mais elevado, poderia levar ao uso de configurações de frequências que garantem resultados sub-ótimos, devido ao critério utilizado para diminuir o número de combinações de frequências exploradas. Como já foi referido na secção anterior, nestes modelos, principalmente se forem considerados um menor número de iterações, a média do tempo de iteração das mesmas poderá facilmente exceder em 20% a referência recolhida devido à grande variação que esta métrica apresenta. No entanto, isto não significa que uma amostra de iterações executadas sobre uma combinação de frequências mais baixas não respeite o limite de degradação de 5%. Assim, o critério usado neste algoritmo para ignorar a exploração de combinações de frequências mais baixas na amostragem é demasiado apertado para este tipo de modelos o que pode levar a que frequências que poderiam ser adequadas e garantir menor consumo energético não sejam exploradas.

Ainda, durante a monitorização, por serem consideradas amostras compostas por 5 iterações, o tempo médio de iteração observado poderia ultrapassar o limite de 5% de degradação sem que isso sig-

nificasse que a combinação de frequências aplicada era inadequada. Assim, seria analisado o desvio padrão que, devido ao **Coeficiente de Variação** elevado destes modelos, teria, com grande probabilidade um valor alto. Por sua vez, o desvio padrão referência deverá assumir um valor baixo. Na primeira iteração, são carregados os primeiros fragmentos do *dataset* (tantos quantos o número de trabalhadores usados para o carregamento de dados) e, por isso, a duração das iterações que se seguem (e que são usadas como referência) deve-se, em grande parte, à computação, pois os dados já se encontram disponíveis. Ainda, a variação no tempo de iteração destes modelos deve-se maioritariamente ao carregamento dos dados, assim, sendo que, nessas iterações, os fragmentos já foram carregados previamente, a sua duração não é afetada por essa variação e, por isso, o desvio padrão referência será um valor menor do que o valor de desvio padrão observado para estes modelos. Pelo que, ao analisar o desvio padrão obtido sobre uma dada configuração, o algoritmo muito provavelmente iria abandonar a combinação de frequências aplicada por a considerar inadequada ainda que pudesse não ser esse o caso. Assim, o uso deste algoritmo não é adequado para os modelos com maior coeficiente de variação do tempo de iteração.

Algorithm 2 Algoritmo para modelos com baixo coeficiente de variação de tempo de iteração.

Inicialização: $samples[5]; queue[1]; freqs = Frequencies(); \omega = 1.05; \gamma = 1.20; \theta = 1.05;$
$train\_data = Data(); val\_data = Data()$

$queue \leftarrow queue.init()$
if min_load_freq **then**
    $freqs.load\_freq = min\_freq$
else
    $freqs.load\_freq = max\_freq$
end if
$freqs.copy\_freq = max\_freq$
$freqs.compute\_freq = max\_freq$
while $True$ **do**
    $it \leftarrow queue.dequeue()$
    if $it.is\_validation()$ **then**
        $data \leftarrow val\_data$
    else
        $data \leftarrow train\_data$
    end if
    if $it.it\_no = 2$ **then**
        $samples.clear()$
    end if
    $samples.push(it)$
    if $data.sampling$ & $samples.full()$ **then**
        $avg\_time, avg\_energy \leftarrow samples.averages()$
        $std\_dev\_time, \_ \leftarrow samples.std\_deviation()$
        if $freqs.freq\_max()$ **then**
            $data.ref\_time = avg\_time$
            $data.ref\_energy = avg\_energy$
            $data.ref\_std\_dev = std\_dev\_time$
        end if

```
28:         if avg_time < ω * data.ref_time & avg_energy ≤ data.ref_energy then
29:             data.topk.add(Frequencies(freqs, avg_time, avg_energy)
30:         end if
31:         if avg_time > γ * data_ref_it_time then
32:             data.skip_freq()
33:         else
34:             freqs ← data.find_next_sampling_freqs()
35:         end if
36:         samples.clear()
37:         if not data.sampling then
38:             data.apply_best_combo()
39:         end if
40:     else if samples.full() then
41:         avg_time, avg_energy ← samples.averages()
42:         std_dev_time, _ ← samples.std_deviation()
43:         if avg_time > ω * data.ref_time then
44:             if freqs.freq_max() then
45:                 data.reset()
46:             else if dev_time <= θ * data.ref_std_dev then
47:                 if not data.topk.try_next() then
48:                     data.reset()
49:                 end if
50:             end if
51:         end if
52:         samples.clear()
53:     end if
54: end while
```

4.3.3 Monitor de Energia e Aplicador de Frequência

As **GPU**s utilizadas nesta dissertação foram **GPU**s NVIDIA pelo que, para a implementação destas duas componentes, foi utilizado o *package* Python pynvml[1] que implementa as funções disponibilizadas pela

[1] https://pypi.org/project/pynvml/

biblioteca NVML da NVIDIA. No entanto, tal como referido anteriormente, pode facilmente adicionar-se suporte para **GPU**s de outras distribuidoras, implementando as funções disponibilizadas no Monitor de Energia e o Aplicador de Frequência de acordo com as ferramentas disponíveis para a monitorização e controlo energético de **GPU**s dessa distribuidora.

O Monitor de Energia deve sempre ter uma função `get_acc_energy()`, que devolve o valor acumulado de energia consumido pela **GPU** desde que o sistema foi ligado. Ainda, deve implementar a função `get_max_freq()`, que devolve o valor de frequência máxima suportado pela **GPU**. Por sua vez, o Aplicador de Frequências deve incluir a função `set_freq(freq)` que define o limite máximo a aplicar à **GPU** e a função `get_min_freq()` que devolve o valor de frequência mínimo suportado pela **GPU**.

Uma instância do Monitor de Energia é responsável pela monitorização de apenas uma **GPU**, assim como uma instância do Aplicador de Frequência é responsável por alterar o limite da frequência apenas de uma **GPU**. Assim, na implementação que recorre ao pynvml, os construtores destas componentes recebem o identificador da **GPU** correspondente e, recorrendo à função `nvmlDeviceGetHandleByIndex(gpu_id)`, obtém um objeto do tipo `nvmlDevice_t` que, na biblioteca NVML, representa um dispositivo, mais especificamente, uma **GPU**. Através deste objeto, é possível fazer diversos tipos de consulta e comandos sobre a **GPU** em questão.

Assim, para a monitorização do consumo energético, feita recorrendo à função `get_acc_energy()`, foi utilizada a função `nvmlDeviceGetTotalEnergyConsumption(device)` que devolve o consumo energético da **GPU** a que se refere o argumento `device`.

No que diz respeito às funções para averiguar qual a frequência máxima e mínima da **GPU**, foi necessário o uso de mais que uma função. As frequências suportadas pela **GPU** dependem da frequência da memória da **GPU** pelo que a função que devolve essa lista de frequências suportadas para a **GPU**, a `nvmlDeviceGetSupportedGraphicsClocks(device, memory_frequency)`, recebe como argumento a frequência da memória para qual se pretende fazer a consulta. Como no contexto deste trabalho não foi manipulada a frequência da memória da **GPU**, esta manteve-se no valor máximo suportado. Assim, foi necessário utilizar primeiro a função `nvmlDeviceGetSupportedMemoryClocks(device)` para obter a frequência máxima da memória da **GPU** (primeiro elemento da lista obtida) e utilizá-la na função que devolve a lista de frequências suportadas para a **GPU**. Os valores de frequência máximo e mínimo para a **GPU** serão o primeiro e último valores da lista obtida, respetivamente.

Por fim, para a aplicação de limites de frequência às **GPU**s, implementada através da função `set_freq`, foi utilizada a função `nvmlDeviceSetGpuLockedClocks(device, min_freq, max_freq)`. A frequência da **GPU** ficam limitada aos valores entre `min_freq` e `max_freq` pelo que o valor utilizado

para `min_freq` é a frequência mínima suportada pela **GPU** (obtida usando `get_min_freq()`) enquanto que o valor utilizado para `max_freq` é o valor de frequência calculado pelo algoritmo. A utilização desta função da biblioteca NVML requer permissões para executar comandos administrativos.

Capítulo 5

Avaliação

A avaliação feita sobre o WAFT tem como objetivo responder às seguintes questões:

1. Será que o WAFT é capaz de diminuir o consumo energético mantendo o desempenho do treino de modelos **AP**?
2. Será que o WAFT é capaz de diminuir o consumo energético sobre diferentes tipos de dispositivos de armazenamento?
3. Será que o WAFT é capaz de diminuir o consumo energético utilizando diferente número de **GPU**s no treino?
4. Como se compara o WAFT com sistemas do estado de arte?
5. Como se comporta o WAFT considerando diferentes limites para a degradação do tempo de execução?

5.1 Ambiente Experimental e Metodologia de Teste

Nesta secção será apresentada a metodologia utilizada na avaliação do WAFT, assim como as cargas de trabalho e métricas recolhidas. São ainda descritos os cenários de teste cujos resultados são expostos nas próximas secções. Os ambientes de teste em que foram realizados os testes são os mesmos utilizados para o estudo descrito no capítulo 3 (detalhados na subsecção 3.1.1).

5.1.1 Cenários de Teste

- **Treino com uma Única GPU.** Este cenário de teste consiste no treino dos modelos utilizando uma única **GPU** com o *dataset* armazenado localmente.

- **Treino com Múltiplas GPUs.** Neste cenário foram utilizadas quatro GPUs no treino dos modelos com o *dataset* num dispositivo de armazenamento local.
- **Treino com o Dataset em Armazenamento Remoto.** Para este cenário o *dataset* foi acedido remotamente durante o treino dos modelos, no qual foi utilizada uma só GPU.
- **Limite de Degradação do Tempo de Execução.** Neste cenário de teste é analisado o comportamento do WAFT quando este sistema considera diferentes limites para a degradação do tempo de execução dos modelos.
- **Treino de Longa Duração.** Neste cenário de teste foram analisados os efeitos do WAFT sobre o treino de um modelo durante um maior número de épocas.

5.1.2 Carga de Trabalho

Os resultados apresentados foram obtidos treinando os modelos AlexNet [19], ResNet18, ResNet101 [15], ShuffleNet (v2 x0.5) [23] e VGG19 [38], recorrendo à aplicação PyTorch [35], com o *dataset* ImageNet [12], que se divide em dois: o *dataset* de treino, correspondente a cerca de 96% do *dataset* completo e o de validação, que corresponde a cerca de 4% (Tabelas 6 e 7). Assim, o treino deste modelo divide-se em épocas, sendo que cada uma se caracteriza por duas fases: o treino e a validação do modelo, sendo a primeira fase mais intensiva computacionalmente. Ainda, o *batch size* utilizado para cada modelo em cada um dos ambientes de teste está detalhado nas Tabelas 6 e 7. Adicionalmente, o critério de paragem utilizado foi a conclusão de três épocas em todos os cenários de teste, à exceção do último, para o qual o número de épocas foi fixado em 45.

Uma vez que estas cargas de trabalho não incluem o pré-processamento dos dados, esta fase não será considerada na avaliação.

Tabela 6: Cargas de Trabalho - *Commodity*.

Modelo	Batch Size	Dataset
AlexNet	128	Imagenet (140 GB / 6.4 GB)
ResNet18	128	
ResNet101	32	
ShuffleNet	128	
VGG19	64	

Tabela 7: Cargas de Trabalho - *Deucalion*.

Modelo	Batch Size	Dataset
AlexNet	256	Imagenet (140 GB / 6.4 GB)
ResNet18	256	
ResNet101	256	
ShuffleNet	256	
VGG19	256	

5.1.3 Métricas

Durante os testes foi continuamente recolhida informação sobre a utilização da(s) **GPU**(s) e da respetiva memória (recorrendo ao `nvidia-smi`), a duração de cada fase da iteração, assim como o consumo energético da(s) **GPU**(s) por iteração. Para além disso, foram monitorizados o consumo energético total relativo à(s) **GPU**(s) e ao tempo de execução global do treino.

5.1.4 Metodologia de teste

Para cada um dos cenários de teste anteriormente descritos, os modelos foram treinados sobre as configurações padrão (sem qualquer limite aplicado à frequência), com o WAFT a controlar a frequência e, por fim, com o GEEPAFS [48], sistema do estado da arte, a controlar a frequência.

Para utilizar o WAFT foi necessário definir o subconjunto de frequências suportadas para serem consideradas pelo algoritmo de controlo (descritas na Tabela 8) e, ainda, utilizar o *hardware profiler* em cada ambiente de teste antes da primeira execução do sistema. Esta componente foi utilizada para descobrir qual o período mínimo de inutilização entre períodos de alta utilização para o qual compensa aplicar a frequência mínima, sendo que o valor obtido para o *Commodity* foi de 60 milissegundos, enquanto que para o *Deucalion* foi 45 milissegundos (mais detalhes sobre estes valores no Apêndice A.1).

Por sua vez, o GEEPAFS foi executado com os argumentos `mod Assure p95`, modo utilizado para garantir que o desempenho se mantém no percentil 95, o que vai de encontro ao limite máximo de 5% de degradação do tempo de execução utilizado por padrão pelo WAFT. Também para este sistema é necessário definir um subconjunto de frequências a utilizar durante a fase de *probing* pelo que foram escolhidos os mesmos subconjuntos utilizados no WAFT (Tabela 8).

Tabela 8: Frequências utilizadas para cada ambiente de teste.

Ambiente de Teste	Frequências (MHz)
Commodity	300, 345, 510, 690, 870, 990, 1125, 1260, 1560, 1710, 1860, 2100
Deucalion	210, 315, 420, 525, 630, 735, 840, 945, 1050, 1155, 1260, 1365, 1410

Durante o treino dos modelos é ainda utilizado o `nvidia-smi` para registar métricas de utilização da(s) **GPU**(s) e da respetiva memória, sendo que esta consulta é feita de segundo a segundo. De acordo com a documentação desta ferramenta, o valor de utilização da **GPU** corresponde à percentagem de tempo em que a mesma foi utilizada e não à intensidade da carga sobre ela imposta, assim como a utilização da memória da **GPU** corresponde à percentagem de tempo em que foram realizadas escritas

ou leituras sobre ela.

Além disso, foi também registada a duração de cada fase em cada iteração, medições feitas através da recolha de *timestamps* no momento da conclusão de cada uma das fases. Assim, a fase de carregamento representa o tempo entre o final da computação da iteração anterior e a conclusão do carregamento do novo fragmento. Deste modo, os valores apresentados para esta fase não incluem o carregamento total do fragmento, apenas o período em que a **GPU** fica inutilizada à espera de dados, ou seja, a porção do carregamento que não se sobrepõe ao final da iteração anterior. Por sua vez, as fases de cópia e computação, tal como os nomes indicam, dizem respeito à duração da cópia da memória da **CPU** para a memória da **GPU** e à computação sobre os dados, respetivamente.

No treino multi-**GPU**, a comunicação entre as várias **GPU**s para sincronização dos gradientes e posterior atualização dos parâmetros do modelo são processos contabilizados no tempo de carregamento do fragmento seguinte. Isto acontece pois estes processos não são bloqueantes, pelo que será registado o final da computação e estes processos continuarão a decorrer. Quando terminarem, será verificado se o carregamento do fragmento seguinte também está concluído e, quando tal se verificar, é registado o término de tal fase. Assim, os valores de duração da fase de carregamento em contexto de treino multi-**GPU** não correspondem totalmente a períodos de inutilização devido à espera por dados pois também contabilizam a sincronização e atualização de gradientes.

Para o treino dos modelos foram utilizados quatro trabalhadores para o carregamento de dados, sendo que cada um carrega um fragmento do *dataset*, pelo que, para dar início à primeira iteração, é necessário esperar que pelo menos um dos trabalhadores termine o carregamento, levando a um longo período de espera por dados nesta iteração. No entanto, para as três iterações seguintes, os fragmentos já estarão disponíveis, uma vez que o seu carregamento terá terminado aproximadamente ao mesmo tempo que o carregamento do fragmento usado na primeira iteração, pelo que a fase de carregamento será muito menor. Assim, nestes modelos existe um ciclo de uma iteração com o período de carregamento mais longo seguida de três iterações com uma fase de carregamento bem menor. Deste modo, os modelos intensivos em movimento de dados deixam que essa variação na fase de carregamento seja refletida no seu tempo de iteração, ao contrário dos modelos intensivos em computação, em que a duração da fase de carregamento dos fragmentos é quase sempre nula, já que o carregamento dos fragmentos se sobrepõe à computação sobre fragmentos anteriores.

Para os resultados apresentados nas próximas secções foi garantido que o ambiente de teste apresentava as mesmas condições. Adicionalmente, os resultados apresentados nas secções 5.2, 5.3 e 5.5 foram calculados através da média dos valores obtidos em três execuções do teste em questão. Por outro

lado, para o cenário de teste apresentado na secção 5.4 foram feitas nove execuções de cada teste pois, nessas condições, os valores apresentam muita variação entre execuções devido ao acesso ao armazenamento remoto. Dessas nove execuções, foram escolhidas três cujo tempo de execução se aproximasse da média. Por fim, os resultados da secção 5.6 foram obtidos através de apenas uma execução do treino do modelo sobre cada configuração.

5.2 Treino com uma única GPU

Apesar de ser aplicável em cenários de treino multi-**GPU**, o WAFT foi desenvolvido com foco em treino numa única **GPU**, pelo que são essas as condições consideradas neste primeiro cenário de teste. Foram feitos testes em diferentes ambientes de teste e sobre diferentes modelos de forma a responder às perguntas inicialmente expostas neste capítulo.

Commodity. Nesta máquina, utilizando o WAFT, foram registadas diminuições do consumo energético entre 15% e 42% incorrendo num aumento máximo do tempo de execução de 6.98%, enquanto que, utilizando o GEEPAFS, foram registadas poupanças de energia entre 7% e 30% com uma degradação máxima do tempo de execução de 9.90% (Figura 26). Nestes resultados não foi considerada a primeira época de treino, já que, durante esse período, o WAFT não faz qualquer ajuste de frequência, apenas o *profiling* do comportamento do modelo. Adicionalmente, em cenários reais, os modelos são treinados durante centenas de épocas, pelo que a primeira época não terá grande impacto no consumo energético e tempo de execução global do treino.

Ainda através da análise da Figura 26, observamos que o GEEPAFS incorre numa maior degradação do tempo de execução nos modelos ResNet101 e VGG19. Como podemos verificar na Figura 27, o que estes dois modelos têm em comum entre si e que os distingue dos restantes é o facto de serem intensivos em computação, não tendo a **GPU** qualquer período de inutilização (identificado como fase de carregamento na Figura 27), o que os torna mais sensíveis à diminuição de frequência da **GPU**. Nestes modelos, quando é diminuída a frequência da **GPU**, a duração das fases de cópia e computação aumenta e, consequentemente, também o tempo de iteração aumenta. No entanto, para os modelos intensivos em movimento de dados (AlexNet, ResNet18 e ShuffleNet - Figura 27), tal não sucede por terem, durante o treino, longos períodos de inutilização da **GPU**. Assim, a diminuição da frequência da **GPU** durante o treino destes modelos leva também ao aumento da duração das fases de cópia e computação, mas apenas provoca um aumento da sobreposição do final de uma iteração com o carregamento do fragmento seguinte. Deste modo, o período de espera por dados da **GPU** (período de inutilização, indicado como

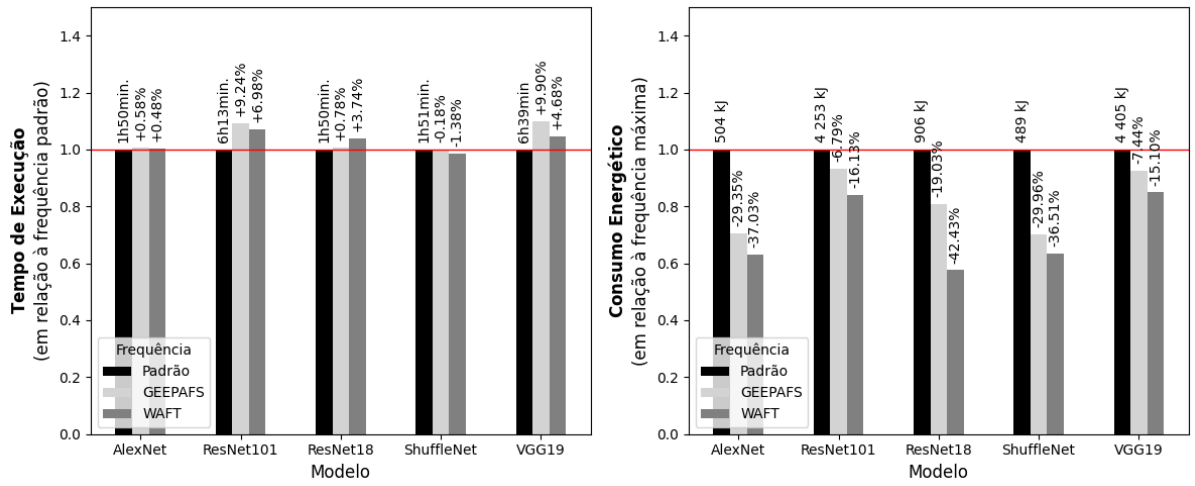


Figura 26: **Tempo de execução e consumo energético da GPU associados ao treino de cada modelo utilizando diferentes algoritmos para a aplicação da frequência (sem a 1ª época do treino).** Experiência no ambiente de teste *Commodity* armazenando o *dataset* localmente.

fase de carregamento na Figura 27) diminui e o tempo de iteração mantém-se. Adicionalmente, nos casos em que o tempo de treino é degradado, a diminuição do consumo energético global deverá ser menor do que a diminuição apresentada para a **GPU**, já que os restantes recursos (p.ex., **CPU**, *cooling*, memória, etc.) estarão a consumir energia durante mais tempo.

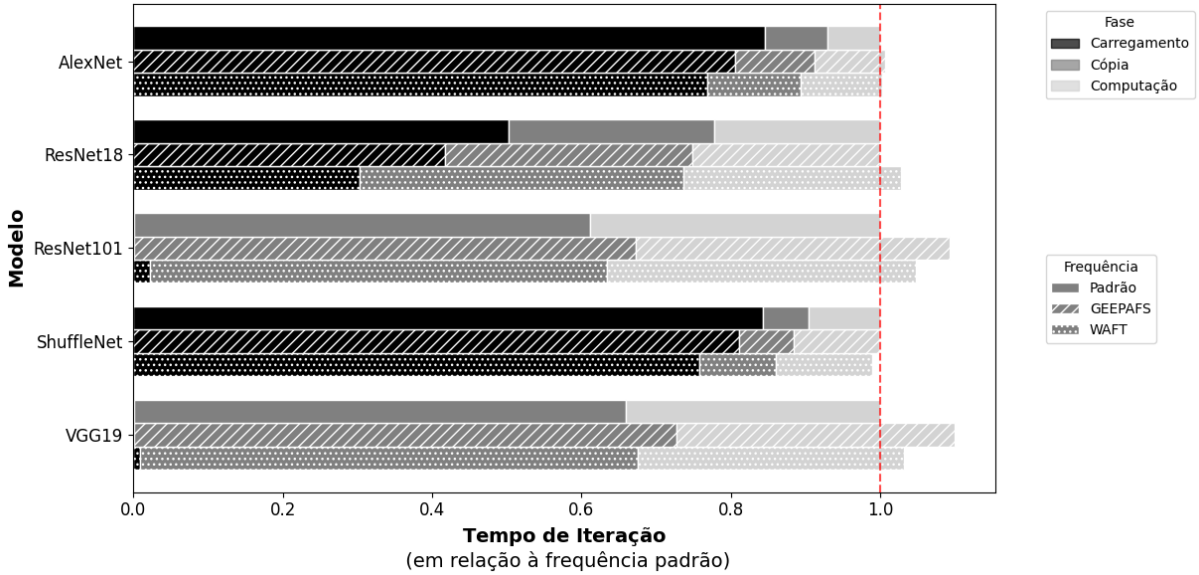


Figura 27: **Duração das fases de iteração por modelo utilizando diferentes algoritmos para a aplicação da frequência**. Experiência no ambiente de teste *Commodity* armazenando o *dataset* localmente.

Analisando ainda a média da utilização da **GPU** durante o treino dos modelos (Figura 28) e tendo em conta que esta representa a percentagem do tempo em que este recurso foi utilizado e não a intensidade da carga que lhe foi imposta, é possível verificar que o WAFT reduziu eficazmente os períodos de inutilização da **GPU** nos modelos em que estes representam grande parte do tempo de treino (AlexNet, ResNet18 e ShuffleNet). Tal é também comprovado pela diminuição da duração da fase de carregamento destes modelos na Figura 27. Isto deve-se ao facto do WAFT ser capaz de identificar os períodos de inutilização da **GPU**, nomeadamente durante o carregamento dos fragmentos, e estender as restantes fases da iteração de forma a minimizar esses períodos. Tendo em conta que os períodos de inutilização para os restantes modelos (ResNet101 e VGG19) são praticamente inexistentes (confirmado pela média do tempo de carregamento na Tabela 9), nesses casos o WAFT apenas alavanca da degradação do tempo de execução para diminuir o consumo energético do treino (Figura 27), pelo que a utilização da **GPU** acaba por diminuir ligeiramente (Figura 28).

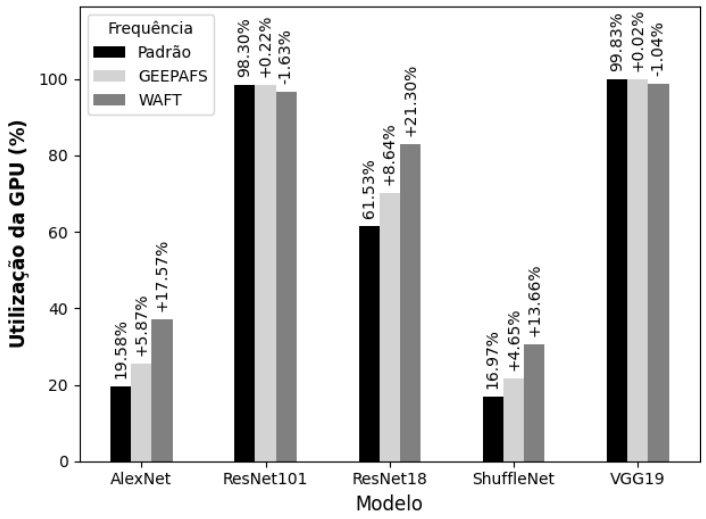


Figura 28: **Utilização da GPU por modelo utilizando diferentes algoritmos para a aplicação da frequência.** Experiência no ambiente de teste *Commodity* armazenando o *dataset* localmente.

Considerando a primeira época, as diminuições do consumo energético, utilizando o WAFT, variaram entre 10% e 28% sem que fosse ultrapassado o limite de 5% de degradação do tempo de execução. Por sua vez, o GEEPAFS alcançou poupanças energéticas entre 7% e 30%, ultrapassando significativamente o limite de 5% da degradação do tempo de execução para os modelos ResNet101 e VGG19 (Figura 29).

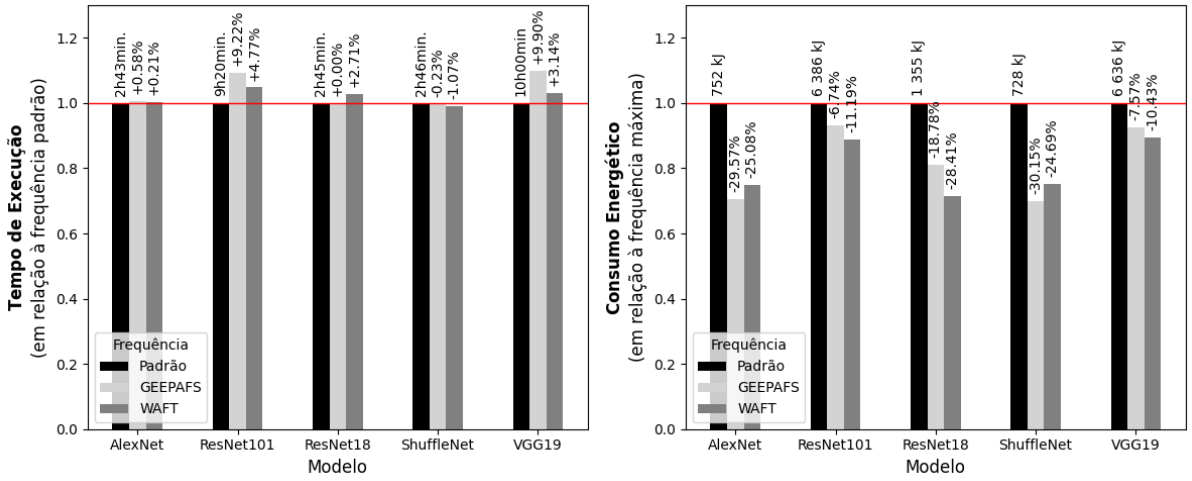


Figura 29: **Tempo de execução e consumo energético da GPU associados ao treino de cada modelo utilizando diferentes algoritmos para a aplicação da frequência.** Experiência no ambiente de teste *Commodity* armazenando o *dataset* localmente.

As decisões tomadas pelo WAFT em termos de algoritmo de controlo aplicado após a primeira época são descritas na Tabela 9, assim como as métricas que foram utilizadas nessa tomada de decisão. Relembrando que é aplicado o algoritmo Alto **CV** a modelos com **Coeficiente de Variação** igual ou superior a 0.5 e Baixo **CV** aos restantes. Ainda, neste ambiente de teste, o período mínimo de inutilização para o qual compensa alterar a frequência para o valor mínimo suportado e que, por isso, define se é ou não aplicada a frequência mínima durante o carregamento é de 60 milissegundos.

Tabela 9: **Algoritmo aplicado pelo WAFT no treino de cada modelo.** Experiência no ambiente de teste *Commodity* armazenando o *dataset* localmente.

Modelo	Coeficiente de Variação do Tempo de Iteração	Média da Duração da Fase de Carregamento	Algoritmo Aplicado	Frequência Mínima no Carregamento
AlexNet	1.22	253.59 ms	Alto CV	X
ResNet18	0.64	468.40 ms	Alto CV	X
ResNet101	0.07	0.13 ms	Baixo CV	
ShuffleNet	1.24	261.70 ms	Alto CV	X
VGG19	0.07	0.21 ms	Baixo CV	

Deucalion. Nesta máquina, o WAFT consegue também alcançar diminuições do consumo energético consideráveis, entre 13% e 24%, com um aumento máximo do tempo de execução de 5.13%. Por sua vez, o GEEPAFS poupa entre 14% e 19% de energia e regista um aumento do tempo de execução máximo de 22.51% (Figura 30). Tal como os testes feitos no ambiente de teste *Commodity*, também nestes o GEEPAFS incorre numa maior degradação do tempo de execução nos modelos ResNet101 e VGG19.

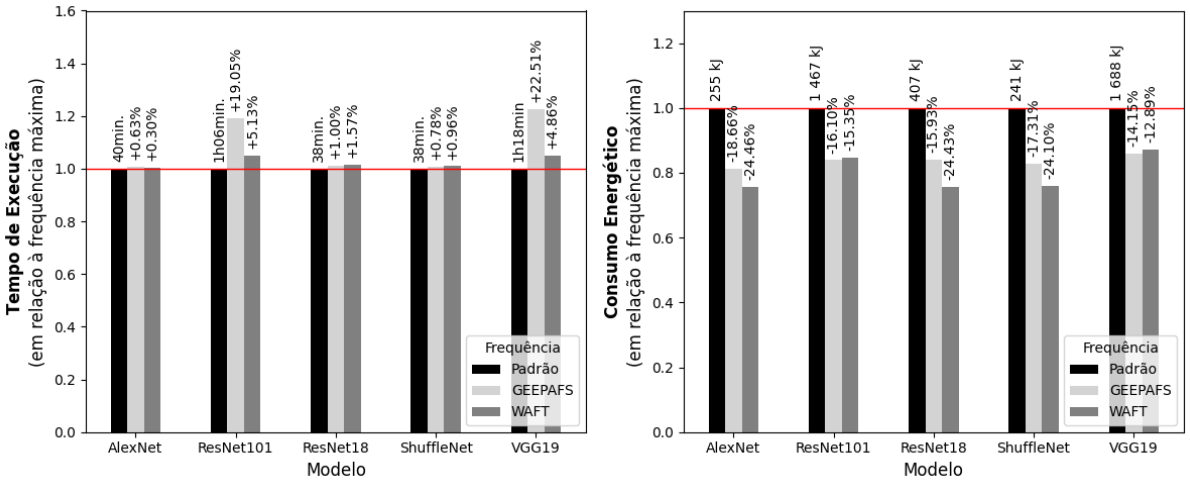


Figura 30: **Tempo de execução e consumo energético da GPU associados ao treino de cada modelo utilizando diferentes algoritmos para a aplicação da frequência (sem a 1ª época).** Experiência no ambiente de teste *Deucalion* utilizando uma **GPU** para o treino e armazenando o *dataset* localmente.

Ainda, observando a distribuição do tempo de iteração por cada fase que a compõe na Figura 31, verificámos que também nestas condições o treino desses modelos é intensivo em computação. Deste modo, a justificação para a degradação expressiva no tempo de execução imposta pelo GEEPAFS no treino destes modelos é a mesma que no ambiente de teste anteriormente estudado.

Adicionalmente, podemos verificar novamente que o WAFT minimiza os períodos de inutilização nos modelos intensivos em movimento de dados (AlexNet, ResNet18 e ShuffleNet) através da extensão das fases de cópia e computação e que, no caso dos modelos intensivos em computação (ResNet101 e VGG19) torna a **GPU** mais lenta para reduzir o consumo energético, sacrificando, para tal, o tempo de iteração (Figura 31).

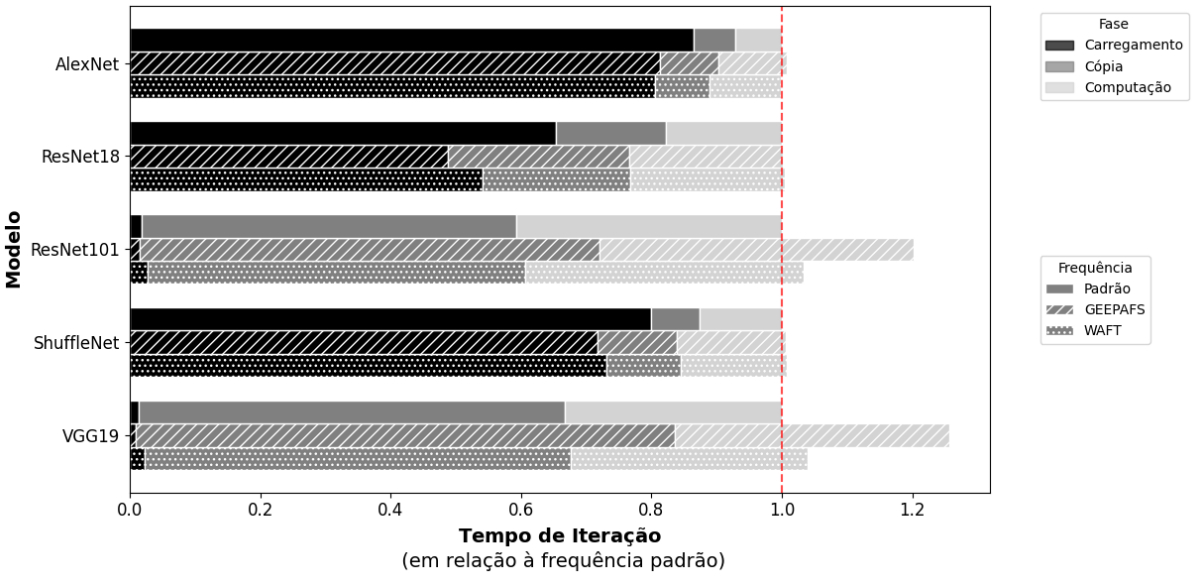


Figura 31: **Duração das fases de iteração por modelo utilizando diferentes algoritmos para a aplicação da frequência.** Experiência no ambiente de teste *Deucalion* utilizando uma **GPU** para o treino e armazenando o *dataset* localmente.

Considerando as métricas relativas à primeira época, observamos que o WAFT obtém poupanças energéticas entre 10% e 17%, não ultrapassando o limite de aumento do tempo de execução de 5% para nenhum dos modelos. Também neste cenário, o GEEPAFS consegue alcançar bons resultados em termos energéticos, registando diminuições entre 15% e 19%, mas incorrendo numa degradação máxima do tempo de execução de 25.66% (Figura 32).

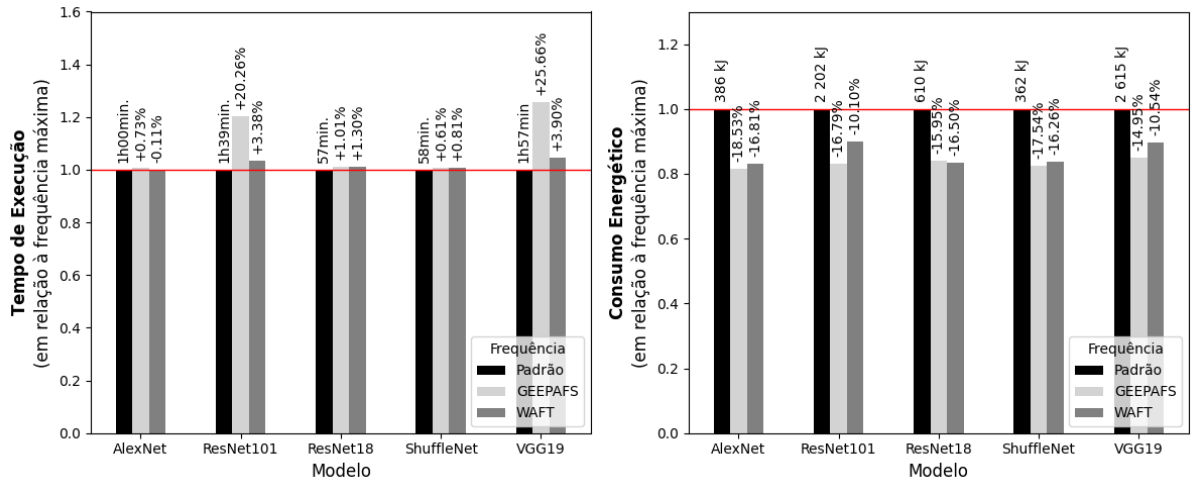


Figura 32: **Tempo de execução e consumo energético da GPU associados ao treino de cada modelo utilizando diferentes algoritmos para a aplicação da frequência.** Experiência no ambiente de teste *Deucalion* utilizando uma **GPU** para o treino e armazenando o *dataset* localmente.

As decisões tomadas pelo WAFT em relação ao algoritmo a aplicar, assim como as métricas nas quais se baseou para tomar a decisão, são apresentadas na Tabela 10. De notar que, para este ambiente de teste, o período mínimo de inutilização para o qual vale a pena alterar a frequência para o valor mínimo é de 35 milissegundos.

Tabela 10: **Algoritmo aplicado pelo WAFT no treino de cada modelo.** Experiência no ambiente de teste *Deucalion* utilizando uma **GPU** para o treino e armazenando o *dataset* localmente.

Modelo	Coeficiente de Variação do Tempo de Iteração	Média da Duração da Fase de Carregamento	Algoritmo Aplicado	Frequência Mínima no Carregamento
AlexNet	1.32	196.92 ms	Alto CV	X
ResNet18	0.91	139.91 ms	Alto CV	X
ResNet101	0.10	0.54 ms	Baixo CV	
ShuffleNet	1.23	174.45 ms	Alto CV	X
VGG19	0.07	0.51 ms	Baixo CV	

Sumário. Este cenário de teste permitiu comprovar que o WAFT é capaz de reduzir o consumo energético com prejuízos no tempo de execução controlados em ambientes de teste diferentes. Ainda, recorrendo à análise do tempo de execução e consumo energético, descartando a primeira época (Figuras 26 e 30), foi possível concluir que, em cenários reais em que os modelos são treinados durante centenas de épocas, o WAFT apresentará melhores resultados do que o GEEPAFS na otimização da eficiência energética, isto é, tanto a nível de poupança energética como a nível da preservação do tempo de execução. Ainda, a diferença dos resultados do GEEPAFS para modelos intensivos em computação e modelos intensivos em movimento de dados realça a necessidade da utilização de algoritmos adaptáveis às características do modelo.

Podemos verificar, através da análise das Tabelas 9 e 10, que o WAFT aplicou o algoritmo Alto CV aos modelos intensivos em movimento de dados (AlexNet, ResNet18 e ShuffleNet - Figuras 27 e 31) e o algoritmo Baixo CV aos modelos intensivos em computação (ResNet101 e VGG19 - Figuras 27 e 31). Isto acontece uma vez que a variação do tempo de iteração nos modelos em treino numa única **GPU** é maioritariamente causada pelo carregamento dos fragmentos. Nos modelos intensivos em movimento de dados, a computação sobre um fragmento termina e é necessário esperar pelos dados para começar

a iteração seguinte, assim, sabendo que este tempo de espera varia, o tempo de iteração irá deixar transparecer esta variação. Por outro lado, nos modelos intensivos em computação, as fases de cópia e computação são suficientemente longas para se sobreporem ao carregamento dos fragmentos, por isso, o início da iteração seguinte depende do fim da computação da anterior e não do fim do carregamento pelo que, variações no tempo de carregamento não afetam o tempo de iteração destes modelos.

5.3 Treino em Múltiplas GPUs

Este cenário de teste tem como objetivo comprovar a eficiência do WAFT em contexto de treino multi-**GPU** e responder às questões inicialmente formuladas.

Deucalion - 4 GPUs. Utilizando as quatro **GPU**s disponíveis nesta máquina para o treino dos modelos, o WAFT foi capaz de atingir poupanças energéticas entre 23% e 36% com um aumento máximo do tempo de execução de 8.62%. Por sua vez, o GEEPAFS conseguiu diminuições no consumo energético entre 18% e 29% com uma degradação máxima do tempo de execução de 4.12% (Figura 33).

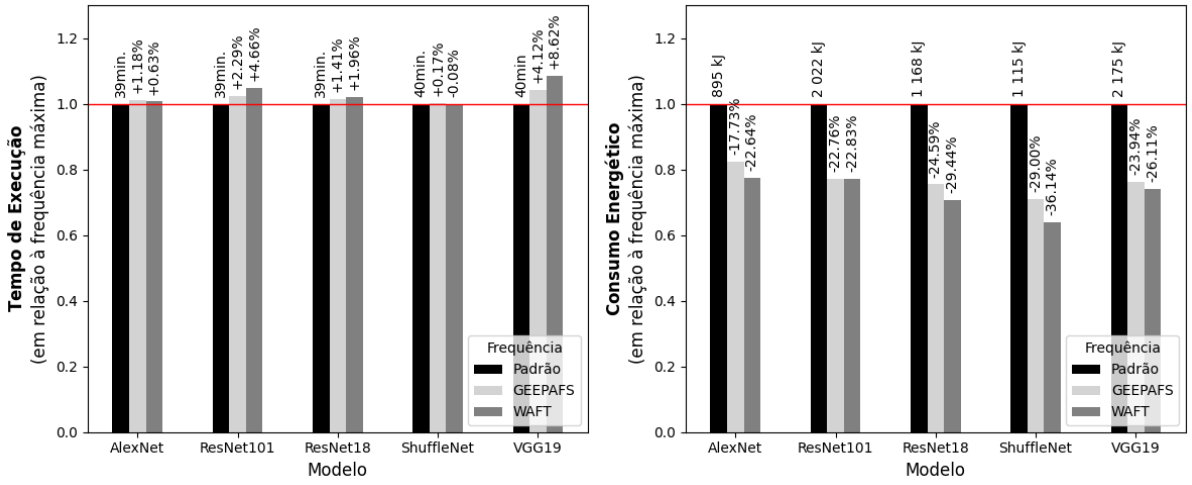


Figura 33: **Tempo de execução e consumo energético das GPUs associados ao treino de cada modelo utilizando diferentes algoritmos para a aplicação da frequência (sem a 1ª época).** Experiência no ambiente de teste *Deucalion* utilizando as quatro **GPU**s para o treino e armazenando o *dataset* localmente.

Como é possível analisar através da Tabela 11, todos os modelos apresentam uma fase de carregamento mais longa do que o período mínimo de inutilização da **GPU** para o qual compensa alterar a frequência para diminuir o consumo energético que, para este ambiente de teste, é de 45 milissegundos,

pelo que o WAFT decidiu aplicar a frequência mínima durante essa fase do treino dos modelos. No entanto, apesar de, em cenários de treino numa única **GPU**, o tempo entre o final da computação e o início da fase de cópia (identificado pelo WAFT como fase de carregamento) ser um período de inutilização, nos cenários de treino multi-**GPU**, não é. Assim, quando o WAFT decide aplicar a frequência mínima durante a fase de carregamento em cenários de treino com múltiplas **GPU**s, pode estar a causar degradação no tempo de execução. Como a comunicação e ajuste dos parâmetros do modelo continuam a acontecer após ter sido registado o fim da computação, por serem não bloqueantes, estes processos são afetados pela frequência mínima quando o WAFT faz tal escolha. Consequentemente, a **GPU** mais lenta torna-se ainda mais lenta, atrasando a sincronização de gradientes para todas as **GPU**s. Ainda, se esta fase for estendida para além do momento em que é concluído o carregamento do fragmento seguinte, leva à degradação do tempo de iteração, pois atrasará o início do processamento desse novo fragmento. Tal pode justificar a degradação mais acentuada do tempo de execução provocada durante o treino do modelo VGG19, já que para este modelo, por ser intensivo em computação (Figura 34), o atraso da sincronização implica o aumento do tempo de iteração.

Tabela 11: **Algoritmo aplicado pelo WAFT no treino de cada modelo.** Experiência no ambiente de teste *Deucalion* recorrendo a 4 **GPU**s e armazenando o *dataset* localmente.

Modelo	Coeficiente de Variação do Tempo de Iteração	Média da Duração da Fase de Carregamento	Algoritmo Aplicado	Frequência Mínima no Carregamento
AlexNet	0.37	103.82 ms	Baixo CV	X
ResNet18	0.38	85.76 ms	Baixo CV	X
ResNet101	0.41	56.24 ms	Baixo CV	X
ShuffleNet	0.36	86.44 ms	Baixo CV	X
VGG19	0.39	76.46 ms	Baixo CV	X

Ainda, o facto da amostragem das combinações de frequência acontecer de forma independente, mas simultaneamente em todas as **GPU**s que participam no treino, também pode ser um entrave para encontrar a frequência ótima para cada **GPU**. O WAFT dá início à fase de amostragem do algoritmo de controlo para cada **GPU**, simultaneamente, no início da segunda época, no entanto, ao alterar a frequência de uma **GPU**, o tempo de iteração das restantes **GPU**s pode ser afetado. Por exemplo,

diminuir a frequência da **GPU** mais lenta irá torná-la ainda mais lenta e irá aumentar o tempo de iteração para todas as **GPU**s, pois terão de esperar mais tempo para a sincronização. Consequentemente, o algoritmo de controlo de cada uma das restantes **GPU**s observa o aumento no tempo de iteração e considera que a frequência por ele aplicada na **GPU** que está a controlar não é adequada, quando essa não é a causa do aumento do tempo de iteração. Por esta razão, em cenários multi-**GPU**, o WAFT pode ter mais dificuldade em encontrar a frequência que é de facto ótima para cada **GPU**.

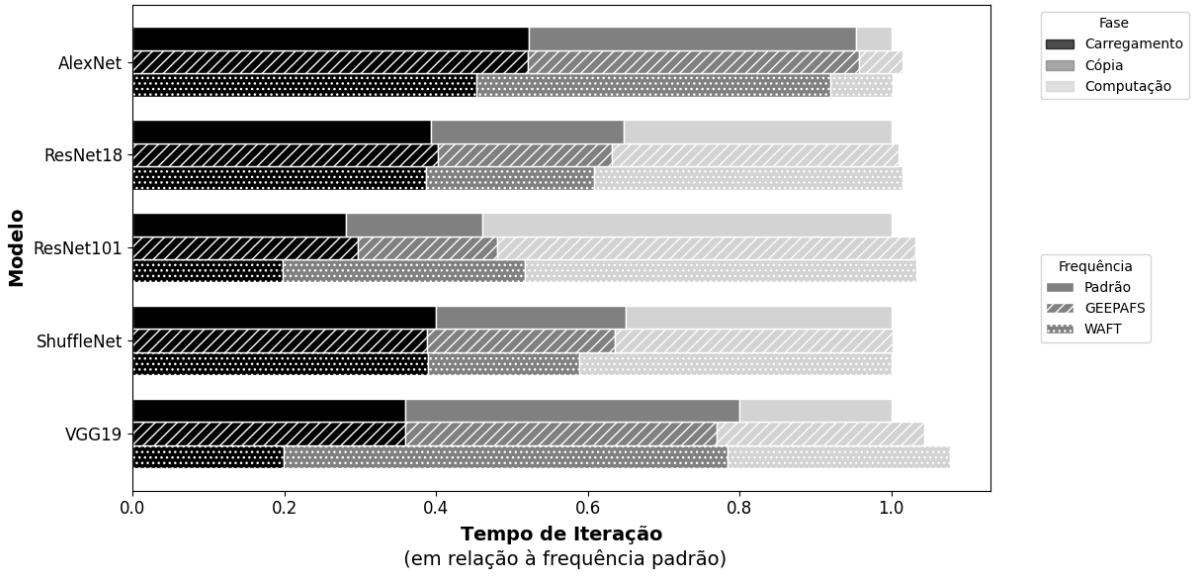


Figura 34: **Duração das fases de iteração por modelo utilizando diferentes algoritmos para a aplicação da frequência.** Experiência no ambiente de teste *Deucalion* utilizando as quatro **GPU**s para o treino e armazenando o *dataset* localmente.

Adicionalmente, através da Figura 35, podemos verificar que o WAFT não foi tão eficiente no aumento da utilização das **GPU**s como no cenário de treino com uma única **GPU**. Tal deve-se ao facto de existirem períodos de inutilização criados devido à sincronização das **GPU**s que o WAFT não identifica como tal e, por isso, não aproveita essas oportunidades de otimização do consumo energético. Durante a sincronização, as **GPU**s mais rápidas têm de esperar pela mais lenta, assim o WAFT poderia baixar a frequência das **GPU**s mais rápidas para que terminassem a sua execução ao mesmo tempo que a **GPU** mais lenta.

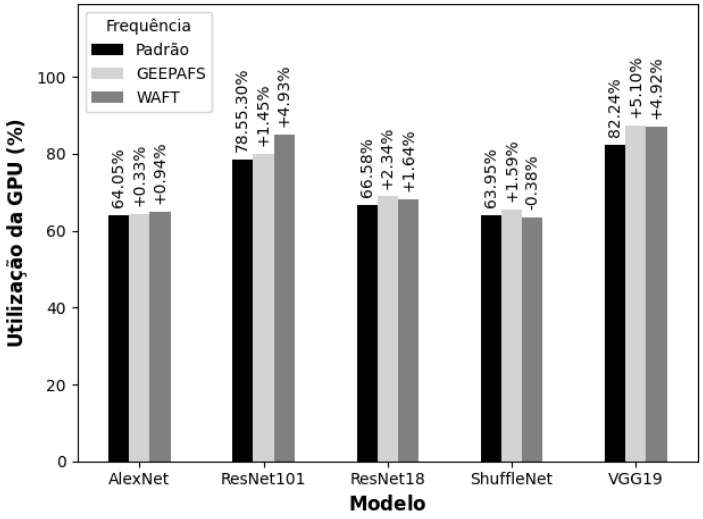


Figura 35: **Utilização média das GPUs por modelo utilizando diferentes algoritmos para a aplicação da frequência.** Experiência no ambiente de teste *Deucalion* utilizando as quatro GPUs para o treino e armazenando o *dataset* localmente.

Analisando também as métricas da primeira época, o WAFT foi capaz de atingir poupanças energéticas entre 15% e 25% com uma degradação máxima do tempo de execução de 5.74%. Por sua vez, o GEEPAFS diminui a energia consumida entre 17% e 28% com um aumento máximo do tempo de execução de 4.42% (Figura 36).

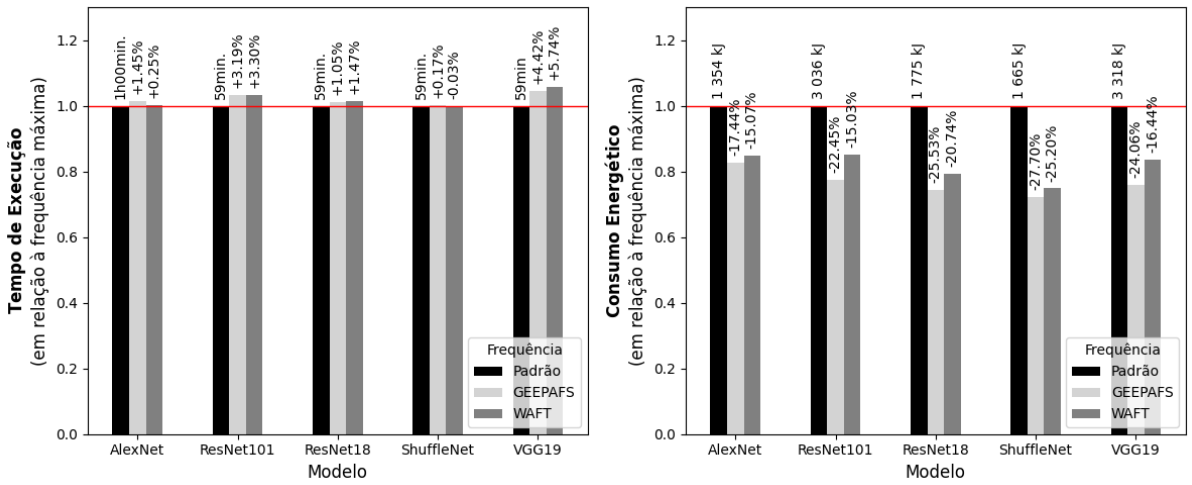


Figura 36: **Tempo de execução e consumo energético das GPUs associados ao treino de cada modelo utilizando diferentes algoritmos para a aplicação da frequência.** Experiência no ambiente de teste *Deucalion* utilizando as quatro GPUs para o treino e armazenando o *dataset* localmente.

Sumário. Este cenário de teste mostrou que o WAFT é também capaz de diminuir consideravelmente o consumo energético no contexto de treino em múltiplas **GPU**s. No entanto, também ficou claro que pode beneficiar da implementação de estratégias que tivessem em conta as particularidades do treino multi-**GPU**. Nomeadamente, a identificação da fase de sincronização e períodos de inutilização criados devido à existência desta fase, o ajuste de frequência para utilizar esses períodos para diminuir o consumo energético e, ainda, a implementação de um método de amostragem de frequências que tome decisões com base na informação agregada de todas as **GPU**s.

Adicionalmente, os resultados registados com o WAFT mostraram que este sistema é, também em cenários de treino em múltiplas **GPU**s, capaz de obter maiores poupanças energéticas do que o GEEPAFS, principalmente em cenários reais em que os modelos são treinados durante centenas de épocas.

5.4 Treino com o Dataset em Armazenamento Remoto

Com o *dataset* armazenado num dispositivo de armazenamento remoto, o treino dos modelos está mais suscetível à existência de um gargalo associado ao carregamento dos dados e, consequentemente, a períodos de inutilização mais longos. Sendo que o foco do WAFT é tirar partido destes períodos para diminuir o consumo energético, foi relevante avaliar a eficiência desta ferramenta neste cenário, respondendo às perguntas inicialmente impostas.

Deucalion. Os resultados obtidos neste cenário foram afetados pelo uso do armazenamento remoto que, por ser partilhado, pode ser acedido, simultaneamente, por outras cargas de trabalho. Assim, o tempo de execução e consumo energético apresentados nesta secção refletem a variabilidade nos acessos a este dispositivo, já que diferentes execuções aconteceram em diferentes momentos e, consequentemente, refletem diferentes níveis de carga na rede que confere acesso ao armazenamento.

Mesmo sujeito à volatilidade de acessos, neste cenário, o WAFT alcançou poupanças energéticas entre 10% e 27% incorrendo num aumento máximo do tempo de execução de 3.81%. Por sua vez, o GEEPAFS alcançou poupanças energéticas entre 15% e 18% com aumento máximo do tempo de execução de 15.73% (Figura 37). Ainda, na Figura 37 é possível verificar que, devido à volatilidade dos acessos ao dispositivo de armazenamento, em alguns modelos, há uma diferença considerável entre o tempo de execução mínimo e máximo registado. Como esta volatilidade afeta diretamente o tempo de carregamento de dados, faz sentido que os modelos mais afetados por esta variação sejam intensivos em movimento de dados (AlexNet e ResNet18 - Figura 38).

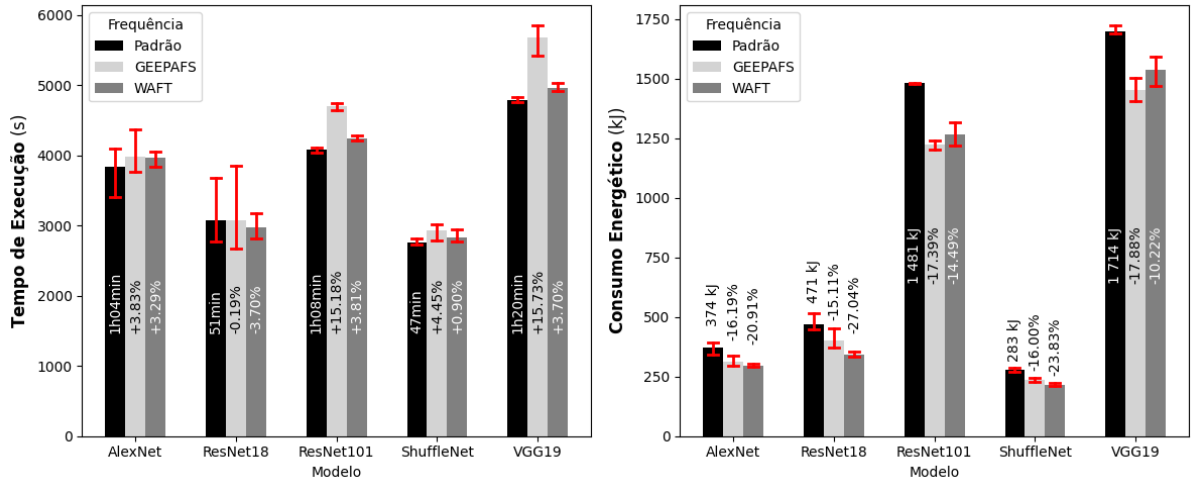


Figura 37: **Tempo de execução e consumo energético da GPU associados ao treino de cada modelo utilizando diferentes algoritmos para a aplicação da frequência (sem a 1ª época).** Experiência no ambiente de teste *Deucalion* utilizando uma **GPU** para o treino e armazenando o *dataset* remotamente.

Tal como nos cenários anteriores, o GEEPAFS apresenta degradações do tempo de execução significativas para modelos intensivos em computação (ResNet101 e VGG19 - Figura 38). Neste caso, podemos supor que estes resultados tenham sido, em parte, afetados pela volatilidade dos acessos ao armazenamento, principalmente para o VGG19, já que foi registada uma grande diferença entre o valor de tempo de execução máximo e mínimo nas três execuções do teste sobre este modelo com o GEEPAFS (Figura 37). No entanto, através da análise do tempo de carregamento não podemos concluir que a volatilidade nas leituras tenha sido a causa exclusiva da degradação do tempo de execução. A duração da fase de carregamento registada no treino do modelo VGG19 quando foi utilizado o GEEPAFS é semelhante à duração registada quando utilizada a frequência padrão (Figura 38), no entanto a duração das restantes fases é bem maior e, por consequência, também o tempo de iteração registado foi mais longo. Todavia, tal comportamento pode dever-se à aplicação de frequências inadequadas ou ainda a um aumento do tempo de carregamento que foi mascarado pelo ajuste de frequências por parte do GEEPAFS, isto é, este ajuste poderia prolongar as fases de cópia e computação aumentando a sobreposição com o carregamento e, consequentemente, o período de inutilização por espera de dados (fase de carregamento na Figura 38) manteria a mesma duração. Deste modo, a degradação do tempo de execução pode dever-se em parte à volatilidade de acessos ao armazenamento, no entanto, também a aplicação de frequências inadequadas por parte do GEEPAFS pode ter contribuído. Este modelo, por ser intensivo em computação, é menos sensível à volatilidade nas leituras e mais sensível à diminuição da frequência e, ainda, pelas conclusões

tiradas nos cenários anteriores, o GEEPAFS não é favorável à preservação do tempo de execução do treino de modelos deste tipo.

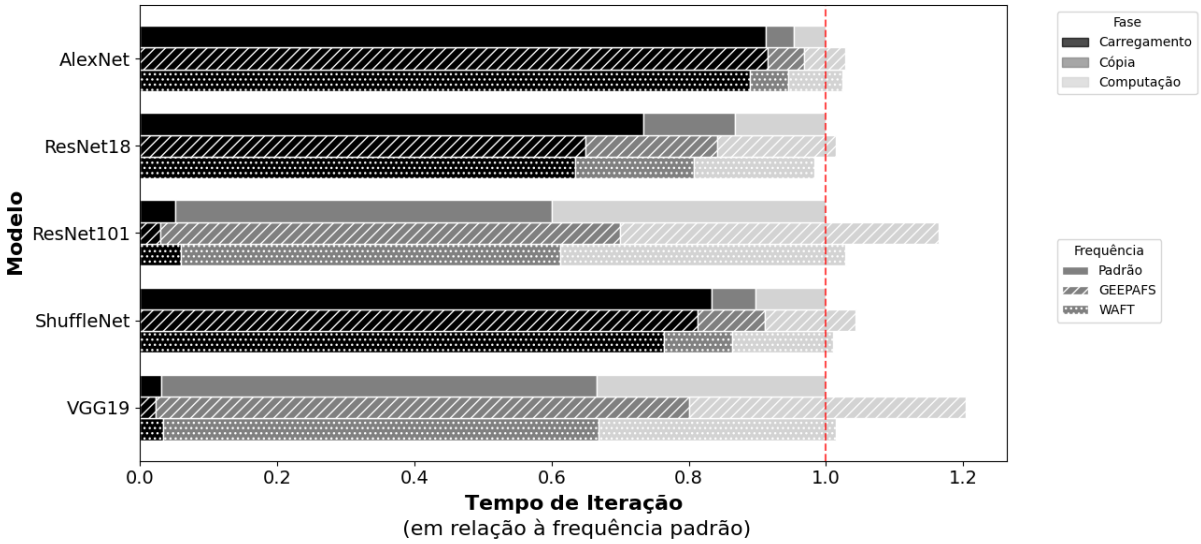


Figura 38: **Duração das fases de iteração por modelo utilizando diferentes algoritmos para a aplicação da frequência.** Experiência no ambiente de teste *Deucalion* utilizando uma **GPU** para o treino e armazenando o *dataset* remotamente.

Analisando as métricas relativas às três épocas executadas, verificamos que WAFT conseguiu poupar entre 7% e 18% com um aumento máximo do consumo energético de 2.85% e o GEEPAFS alcançou poupanças energéticas entre 14% e 17% incorrendo numa degradação máxima do tempo de execução de 33%.

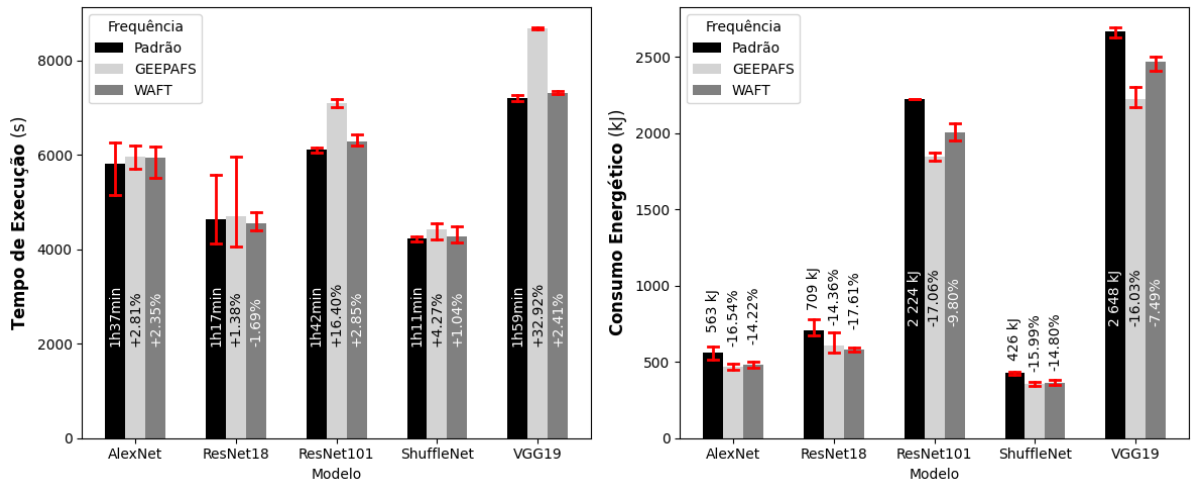


Figura 39: **Tempo de execução e consumo energético da GPU associados ao treino de cada modelo utilizando diferentes algoritmos para a aplicação da frequência.** Experiência no ambiente de teste *Deucalion* utilizando uma **GPU** para o treino e armazenando o *dataset* remotamente.

Sumário. Este cenário de teste permitiu concluir que o WAFT, não podendo escapar à volatilidade introduzida pelos acessos ao dispositivo de armazenamento remoto, alcança valores de poupança energética significativos. Adicionalmente, neste cenário, o WAFT prova conseguir obter melhores resultados do que o GEEPAFS, já que é capaz de preservar o tempo de execução dos modelos (tendo em conta a disponibilidade e desempenho dos recursos) e, ainda assim, alcançar poupanças energéticas consideráveis.

5.5 Limite de Degradação do Tempo de Execução

Neste cenário foi alterado o parâmetro utilizado pelo WAFT para definir o limite de degradação do tempo de execução. Adicionalmente, foi necessário ajustar o parâmetro que define se compensa, durante o processo de amostragem, prosseguir a exploração para frequências mais baixas ou se devem ser ignoradas por ser certo que não respeitarão o limite de degradação do tempo de execução.

Commodity. Neste ambiente de teste, foram feitos testes aplicando um limite máximo à degradação do tempo de execução de 10% e 15%. Quando utilizado o limite de 10%, a amostragem no algoritmo Baixo **CV**, ao experimentar uma combinação de frequências que exceda o tempo de iteração de referência em mais 30%, irá ignorar todas as combinações que consistam em frequências mais baixas, já no algoritmo Alto **CV** a percentagem utilizada foi de 60%. Por outro lado, quando o limite à degradação do tempo de execução foi fixado em 15%, no algoritmo Baixo **CV**, são ignoradas combinações de frequências mais

baixas quando uma o tempo de iteração de uma combinação excede em 35% a referência para essa métrica e, no algoritmo Alto **CV**, tal sucede, ao ultrapassar os 65%.

Definindo o limite de degradação do tempo de execução como 10%, o WAFT conseguiu poupanças energéticas entre 24% e 43% com um aumento máximo do tempo de execução de 10.24%. Por outro lado, quando o limite máximo de aumento do tempo de execução foi fixado em 15%, o WAFT diminuiu o consumo energético entre 25% e 43%, incorrendo num aumento máximo do tempo de execução de 12.25% (Figura 40).

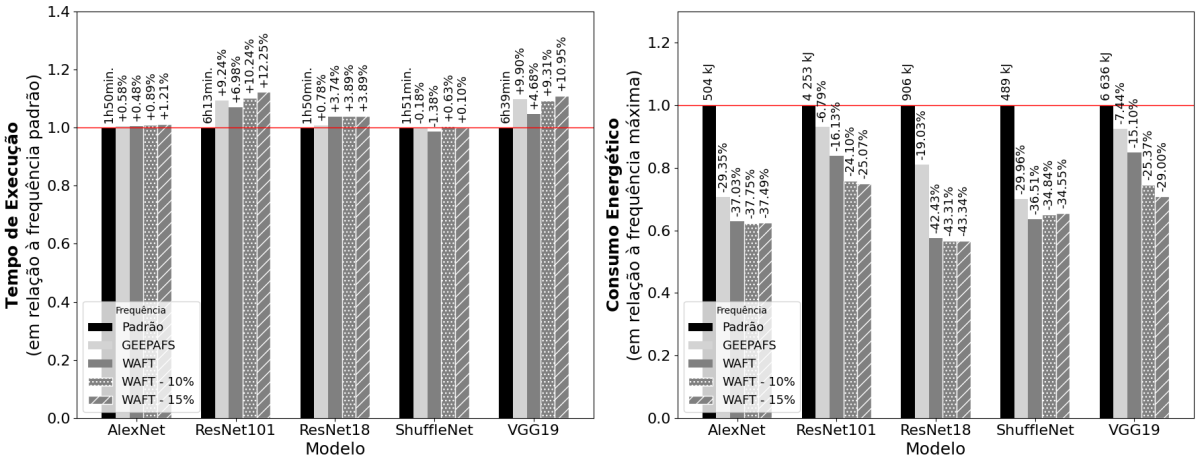


Figura 40: **Tempo de execução e consumo energético da GPU associados ao treino de cada modelo utilizando diferentes algoritmos para a aplicação da frequência (sem a 1ª época).** Experiência no ambiente de teste *Commodity* armazenando o *dataset* localmente.

Os efeitos da utilização de diferentes limites para a degradação do tempo de execução são visíveis principalmente nos modelos intensivos em computação (ResNet101 e VGG19). Como estes modelos não têm períodos de inutilização durante a iteração devido à espera por dados, a única forma de diminuir o seu consumo energético é diminuindo a frequência de operação, aumentando, consequentemente, o tempo de iteração. Assim, nestes casos, o WAFT consegue otimizar apenas através da degradação do tempo de execução, por isso, aumentando o limite definido para essa degradação, é possível aumentar a percentagem de diminuição de consumo energético.

No entanto, para os modelos intensivos em movimento de dados, pode ter sido atingida a configuração que garante o menor consumo energético mesmo com o limite de degradação do tempo de execução fixado em 5%. Como foi demonstrado no Estudo Motivacional pelas Figuras 20 e 21, não é garantido que um aumento maior no tempo de iteração signifique que a energia consumida na execução da mesma seja menor, pelo que, aplicar uma configuração que degrade mais o tempo de execução destes modelos pode levar a um consumo energético maior do que o obtido com pequenas percentagens de degradação.

Considerando também as métricas da primeira época, com um limite de degradação do tempo de execução máximo de 10%, o WAFT diminui o consumo energético entre 16% e 29% incorrendo num aumento máximo do tempo de execução de 6.90%. Já quando o limite é fixado em 15%, o WAFT obteve poupanças energéticas entre 17% e 29% com um aumento máximo do tempo de execução de 8.28%.

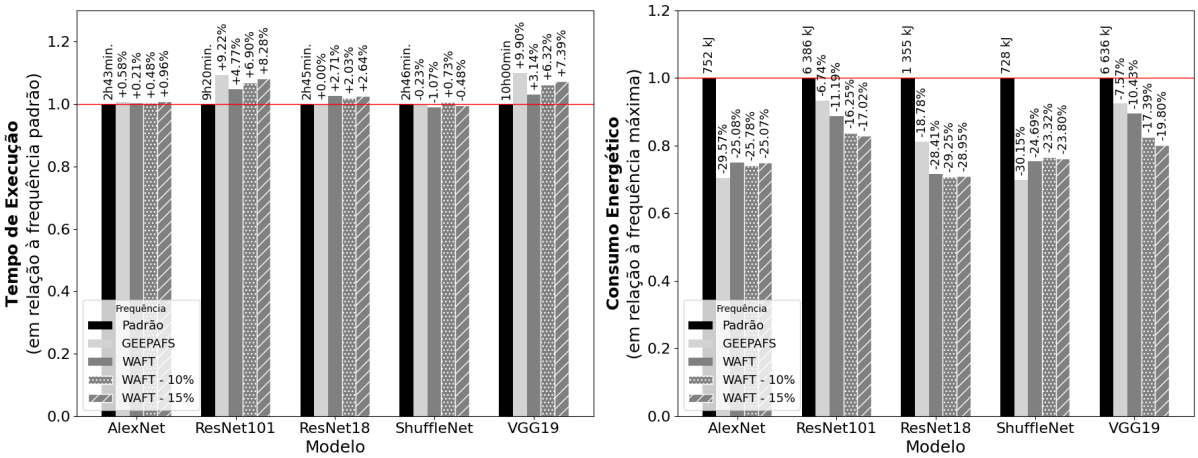


Figura 41: **Tempo de execução e consumo energético da GPU associados ao treino de cada modelo utilizando diferentes algoritmos para a aplicação da frequência.** Experiência no ambiente de teste *Commodity* armazenando o *dataset* localmente.

Sumário. Definindo limites mais altos para a degradação do tempo de execução, foi possível alcançar maiores poupanças energéticas para os modelos intensivos em computação. Tal não se verificou para os modelos intensivos em movimento de dados, pois foi possível alcançar a configuração que garante menor consumo energético com o limite de degradação máxima do tempo de execução de 5%.

5.6 Treino de Longa Duração

Para este cenário de teste, foi treinado o modelo ResNet101 durante 45 épocas de forma a avaliar o comportamento do WAFT sobre treinos de longa duração.

Deucalion. Para a apresentação dos resultados foi considerado o maior número de épocas concluídas em aproximadamente 24 horas com cada uma das configurações de frequência. Enquanto que, com a configuração padrão foram completas 44 épocas consumindo um total de 32 157 kJ, utilizando o WAFT foram concluídas 42 épocas com um consumo energético 17% menor. No entanto, para completar as 45 épocas, o consumo energético previsto seria de 27 907 kJ, apresentando assim uma diminuição de 13% em relação à frequência padrão (Tabela 12).

Tabela 12: **Tempo de execução, consumo energético e acurácia utilizando diferentes algoritmos para a aplicação da frequência.** Experiência no ambiente de teste *Deucalion* utilizando uma **GPU** para o treino e armazenando o *dataset* localmente.

Frequência	Número de Épocas	Tempo de Execução	Consumo Energético	Acurária	Previsão Energética
Padrão	44	24h16min	32 157 kJ	71.45%	
WAFT	42	24h17min	26 638 kJ (-17.16%)	71.85%	27 907 kJ (-13.22%)

Ainda, a acurácia registada no final das 45 épocas com a frequência padrão é semelhante à registada ao fim de 42 épocas de treino sobre as configurações do WAFT (Tabela 12). No entanto, através da análise do gráfico na Figura 42, concluímos que se deve ao facto da acurácia estagnar a partir da época 31.

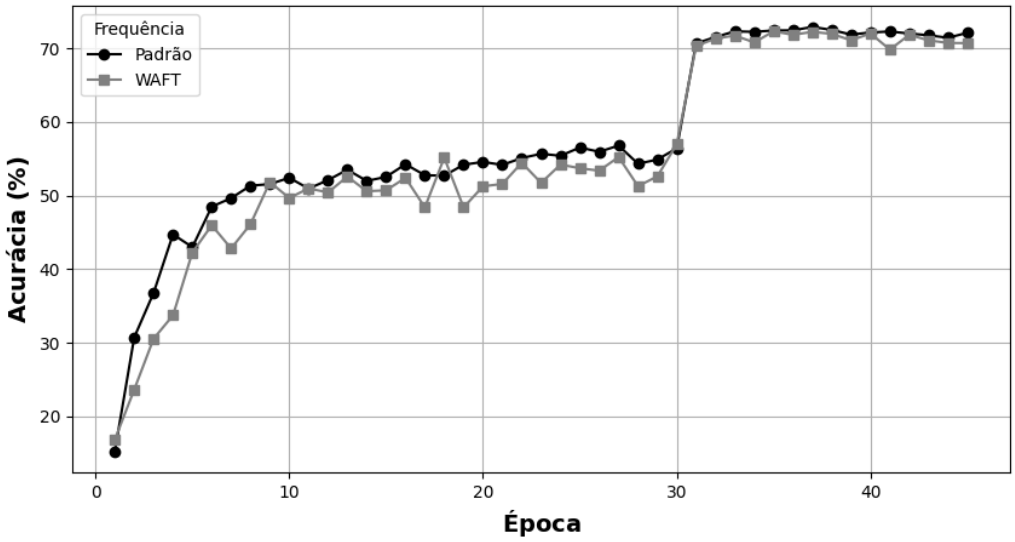


Figura 42: **Acurácia ao longo das épocas utilizando diferentes algoritmos para a aplicação da frequência.** Experiência no ambiente de teste *Commodity* armazenando o *dataset* localmente.

Sumário. Este cenário de teste comprova que, quanto mais longo o treino do modelo, melhores serão os resultados do WAFT e mais aproximados dos valores apresentados nos cenários anteriores em que foi descartada a primeira época. Isto é comprovado pelo facto de, no cenário de teste apresentado na secção 5.2, em que este modelo foi treinado durante apenas 3 épocas, ter sido registada uma diminuição do consumo energético de 10.10%, no entanto, descartando as métricas relativas à primeira época, a poupança energética ter sido de 15.35%.

5.7 Sumário

A avaliação experimental demonstrou a eficiência do WAFT na otimização do consumo energético do treino de modelos de **AP** em diferentes cenários.

Os cenários de teste apresentados nas secções 5.2 e 5.4 demonstraram que o WAFT é capaz de diminuir consideravelmente (até 42%) o consumo energético do treino de modelos em uma única **GPU** preservando o seu tempo de execução inicial em diferentes ambientes de teste e sobre diferentes dispositivos de armazenamento. Adicionalmente, o WAFT provou ser capaz de alcançar resultados melhores que o GEEPAFS (de 7% a 30%), por adaptar o seu comportamento de acordo com as características do modelo. Para além disso, este sistema foi capaz de diminuir os períodos de inutilização da **GPU** através da alteração da frequência da **GPU**, de acordo com a fase da iteração e a carga por ela imposta na **GPU**.

Por outro lado e apesar de não ter sido desenvolvido com foco nos cenários de treino multi-**GPU**, no cenário apresentado na secção 5.3, o WAFT apresenta resultados que revelam diminuições do consumo energético consideráveis (até 36%) mas que demonstram a necessidade da implementação de estratégias que considerem as particularidades destes cenários. Nomeadamente, a criação de algoritmos adaptados a este cenário que recorram à agregação de métricas sobre todas as **GPU**s que participem do treino de forma a serem tomadas decisões melhores e mais informadas. Ou ainda a identificação de novas oportunidades de otimização do consumo energético derivadas da fase de sincronização de gradientes entre as **GPU**s, que provoca períodos de inutilização em alguns dispositivos por terem de esperar pelo mais lento.

Ainda, no cenário de teste apresentado na secção 5.5, o WAFT mostra ser capaz de alcançar diferentes valores de poupança do consumo energético de acordo com a importância que o utilizador dá à degradação do tempo de execução.

Por fim, o último cenário de teste (secção 5.6) demonstrou que o WAFT também consegue diminuir o consumo energético de treinos de maior duração, inclusive, quanto mais longo for o treino de um modelo, maior percentagem de energia o WAFT conseguirá poupar. Assim prova que, em cenários reais, em que os modelos são treinados durante centenas de épocas, irá obter diminuições do consumo energético consideráveis e obterá melhores resultados do que o GEEPAFS.

Com base nestes resultados, podemos concluir que o WAFT é uma solução viável para a otimização do consumo energético de **GPU**s em contexto de **AP**, contribuindo para a diminuição do consumo energético de tais tarefas, tendo em conta as suas características e as diferentes fases que as compõem, sendo, por isso, capaz de conservar o seu tempo de execução.

Capítulo 6

Conclusões e trabalho futuro

A preocupação com o consumo energético associado ao treino de modelos de **AP** têm crescido com a consciencialização da quantidade de energia consumida nas infraestruturas dedicadas a estes tipos de tarefas. Apesar de terem surgido algumas ferramentas que tentam diminuir o consumo energético neste contexto sem prejudicar o tempo de execução destes processos, não existe ainda nenhuma solução que identifique as diferentes fases de uma iteração e aproveite os períodos de inutilização para melhorar a eficiência energética do treino de modelos. As soluções existentes utilizam o mesmo algoritmo durante o treino de diferentes modelos, aplicando a mesma frequência durante todas as fases de uma iteração. No entanto, o estudo realizado mostra que diferentes fases de uma iteração reagem de forma diferente à aplicação de uma mesma frequência. Adicionalmente, demonstrou que o mesmo se verifica para o treino de diferentes modelos, sendo uns mais sensíveis à diminuição de frequência do que outros, pelo que as suas características devem ser tidas em conta nos algoritmos de otimização energética.

Para responder a estes desafios, esta dissertação apresenta o WAFT, um novo sistema de controlo energético adaptável e de granularidade fina. Esta solução é capaz de identificar corretamente as fases de uma iteração no treino de modelos numa única **GPU** e, tendo em conta, essas fases, a carga que impõem sobre a **GPU** e as características do modelo em questão, encontra a frequência ótima para cada uma das fases. Ainda, por monitorizar continuamente métricas de desempenho do treino do modelo, é capaz de identificar o surgimento de gargalos e adaptar o seu comportamento de acordo com o novo cenário.

Este sistema foi testado através do treino de vários modelos em diferentes ambientes de teste, sobre diferentes dispositivos de armazenamento e recorrendo a diferentes número de **GPU**s. Estas experiências foram conduzidas de forma a comprovar a eficiência do WAFT no que diz respeito à diminuição do consumo energético e preservação do tempo de execução. Esta avaliação permitiu concluir que o WAFT é capaz de alcançar diminuições no consumo energético até 42% quando comparado com o cenário utilizado em ambientes de produção. Adicionalmente, quando comparado com soluções do estado da arte,

o WAFT apresentou poupanças energéticas até 29%. Assim, esta avaliação permitiu concluir que o WAFT é uma solução viável para a otimização do consumo energético de **GPU**s em contexto de **AP**, garantindo maior eficiência energética sem sacrificar o tempo de execução do treino dos modelos.

6.1 Perspetiva de trabalho futuro

O trabalho realizado nesta dissertação abre diferentes caminhos que podem ser explorados como trabalho futuro.

Implementar estratégias adaptadas a treino multi-GPU. Apesar de ter apresentado já resultados promissores em cenários de treino multi-**GPU**, este é um ponto que pode ser melhorado. Poderiam ser implementadas estratégias que considerem as particularidades destes cenários, nomeadamente a identificação da fase de sincronização das **GPU**s e períodos de inutilização criados devido à existência desta fase, a manipulação da frequência para minimizar esses períodos e a implementação de um algoritmo que agregue as métricas relativas a todas as **GPU**s que participam no treino

Adaptar o sistema para utilização em cenários de treino distribuído. Para além de considerar este cenário, seria também interessante considerar o treino distribuído por vários nós. Tal como nos cenários de treino multi-**GPU**, também nestes cenários se poderia tirar proveito dos períodos de inutilização provocados pela sincronização entre **GPU**s.

Expandir o domínio de recursos controlados. Outro caminho de grande relevo seria considerar o controlo de frequência de outros recursos na infraestrutura como a **CPU** ou a memória principal. Apesar do consumo energético do treino do modelos ser maioritariamente da responsabilidade da **GPU**, os restantes recursos de um servidor também participam neste processo e, consequentemente, contribuem para o consumo energético do mesmo. Assim, o ajuste da frequência dos restantes recursos, podia permitir alcançar poupanças energéticas ainda maiores.

Bibliografia

[1] PAPI: Performance Application Programming Interface. https://icl.utk.edu/papi/.

[2] Naman Agarwal, Rohan Anil, Tomer Koren, Kunal Talwar, and Cyril Zhang. Stochastic Optimization with Laggard Data Pipelines. In *Advances in Neural Information Processing Systems*, volume 33, pages 10282–10293. Curran Associates, Inc., 2020. URL https://proceedings.neurips.cc/paper_files/paper/2020/file/74dbd1111727a31a2b825d615d80b2e7-Paper.pdf.

[3] AMD. AMD System Management Interface (AMD SMI) Library - AMD SMI 23.4.0.0 Documentation. https://rocm.docs.amd.com/projects/amdsmi/en/latest/index.html#, .

[4] AMD. AMD ROCm documentation - ROCm documentation. https://rocm.docs.amd.com/en/latest/index.html, .

[5] AMD. ROCm System Management Interface (ROCm SMI) Library - ROCm SMI LIB. https://rocm.docs.amd.com/projects/rocm_smi_lib/en/latest/index.html, .

[6] Pragati Baheti. Train Test Validation Split: How To Best Practices [2024]. 2021. URL https://www.v7labs.com/blog/train-validation-test-set.

[7] Chris M Bishop. Neural networks and their applications. *Review of scientific instruments*, 65(6): 1803–1832, 1994.

[8] Sangjin Choi, Inhoe Koo, Jeongseob Ahn, Myeongjae Jeon, and Youngjin Kwon. EnvPipe: Performance-preserving DNN Training Framework for Saving Energy. In *2023 USENIX Annual Technical Conference (USENIX ATC 23)*, pages 851–864, Boston, MA, July 2023. USENIX Association. ISBN 978-1-939133-35-9. URL https://www.usenix.org/conference/atc23/presentation/choi.

[9] Jae-Won Chung, Yile Gu, Insu Jang, Luoxi Meng, Nikhil Bansal, and Mosharaf Chowdhury. Reducing Energy Bloat in Large Model Training. In *Proceedings of the ACM SIGOPS 30th Symposium on*

Operating Systems Principles, SOSP '24, page 144–159, New York, NY, USA, 2024. Association for Computing Machinery. ISBN 9798400712517. doi: 10.1145/3694715.3695970. URL https://doi.org/10.1145/3694715.3695970.

[10] Paulo Cortez and José Neves. Redes neuronais artificiais. *Unidade de Ensino, Departamento de Informática, Escola de Engenharia, Universidade do Minho, Braga, Portugal*, 2000.

[11] Marco Dantas, Diogo Leitao, Peter Cui, Ricardo Macedo, Xinlian Liu, Weijia Xu, and Joao Paulo. Accelerating Deep Learning Training Through Transparent Storage Tiering. In *2022 22nd IEEE International Symposium on Cluster, Cloud and Internet Computing (CCGrid)*, pages 21–30. IEEE, 2022.

[12] Jia Deng, Wei Dong, Richard Socher, Li-Jia Li, Kai Li, and Li Fei-Fei. Imagenet: A Large-Scale Hierarchical Image Database. In *2009 IEEE conference on computer vision and pattern recognition*, pages 248–255. IEEE, 2009.

[13] Diandian Gu, Xintong Xie, Gang Huang, Xin Jin, and Xuanzhe Liu. Energy-Efficient GPU Clusters Scheduling for Deep Learning. *arXiv preprint arXiv:2304.06381*, 2023.

[14] Sayed Hadi Hashemi, Sangeetha Abdu Jyothi, and Roy Campbell. TicTac: Accelerating distributed deep learning with communication scheduling. *Proceedings of Machine Learning and Systems*, 1: 418–430, 2019.

[15] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep Residual Learning for Image Recognition. In *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 770–778, 2016. doi: 10.1109/CVPR.2016.90.

[16] Zhihao Jia, James Thomas, Todd Warszawski, Mingyu Gao, Matei Zaharia, and Alex Aiken. Optimizing DNN Computation with Relaxed Graph Substitutions. *Proceedings of Machine Learning and Systems*, 1:27–39, 2019.

[17] Zhihao Jia, Matei Zaharia, and Alex Aiken. Beyond Data and Model Parallelism for Deep Neural Networks. In A. Talwalkar, V. Smith, and M. Zaharia, editors, *Proceedings of Machine Learning and Systems*, volume 1, pages 1–13, 2019. URL https://proceedings.mlsys.org/paper_files/paper/2019/file/b422680f3db0986ddd7f8f126baaf0fa-Paper.pdf.

[18] Moez Krichen. Convolutional Neural Networks: A Survey. *Computers*, 12(8), 2023. ISSN 2073-431X. doi: 10.3390/computers12080151. URL https://www.mdpi.com/2073-431X/12/8/151.

[19] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E. Hinton. ImageNet Classification with Deep Convolutional Neural Networks. *Commun. ACM*, 60(6):84–90, May 2017. ISSN 0001-0782. doi: 10.1145/3065386. URL https://doi.org/10.1145/3065386.

[20] Michael Kuchnik, Ana Klimovic, Jiri Simsa, Virginia Smith, and George Amvrosiadis. Plumber: Diagnosing and Removing Performance Bottlenecks in Machine Learning Data Pipelines. *Proceedings of Machine Learning and Systems*, 4:33–51, 2022.

[21] Guillaume Leclerc, Andrew Ilyas, Logan Engstrom, Sung Min Park, Hadi Salman, and Aleksander Mądry. FFCV: Accelerating Training by Removing Data Bottlenecks. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 12011–12020, 2023.

[22] Weile Luo, Ruibo Fan, Zeyu Li, Dayou Du, Qiang Wang, and Xiaowen Chu. Benchmarking and Dissecting the Nvidia Hopper GPU Architecture. *arXiv preprint arXiv:2402.13499*, 2024.

[23] Ningning Ma, Xiangyu Zhang, Hai-Tao Zheng, and Jian Sun. Shufflenet v2: Practical guidelines for efficient CNN architecture design. In *Proceedings of the European Conference on Computer Vision (ECCV)*, pages 116–131, 2018.

[24] Francisco Mendes, Pedro Tomás, and Nuno Roma. Decoupling GPGPU voltage-frequency scaling for deep-learning applications. *Journal of Parallel and Distributed Computing*, 165:32–51, 2022. ISSN 0743-7315. doi: https://doi.org/10.1016/j.jpdc.2022.03.004. URL https://www.sciencedirect.com/science/article/pii/S0743731522000624.

[25] Jayashree Mohan, Amar Phanishayee, Ashish Raniwala, and Vijay Chidambaram. Analyzing and mitigating data stalls in DNN training. *Proc. VLDB Endow.*, 14(5):771–784, 2021. doi: 10.14778/3446095.3446100. URL http://www.vldb.org/pvldb/vol14/p771-mohan.pdf.

[26] Derek G Murray, Jiri Simsa, Ana Klimovic, and Ihor Indyk. tf.data: A Machine Learning Data Processing Framework. *arXiv preprint arXiv:2101.12127*, 2021.

[27] NVIDIA. NVDIA CUDA deep neural network library (cuDNN). https://developer.nvidia.com/cudnn, .

[28] NVIDIA. NVDIA developer data loading library. https://developer.nvidia.com/dali, .

[29] NVIDIA. NVDIA collective communications library (NCCL). https://developer.nvidia.com/nccl, .

[30] NVIDIA. NVIDIA management library. https://developer.nvidia.com/nvidia-management-library-nvml, .

[31] Cristobal Ortega, Lluc Alvarez, Alper Buyuktosunoglu, Ramon Bertran, Todd Rosedahl, Pradip Bose, and Miquel Moreto. Adaptive Power Shifting for Power-Constrained Heterogeneous Systems. *IEEE Transactions on Computers*, 72(3):627–640, 2022.

[32] Sangyoung Park, Jaehyun Park, Donghwa Shin, Yanzhi Wang, Qing Xie, Massoud Pedram, and Naehyuck Chang. Accurate Modeling of the Delay and Energy Overhead of Dynamic Voltage and Frequency Scaling in Modern Microprocessors. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 32(5):695–708, 2013.

[33] Pratyush Patel, Esha Choukse, Chaojie Zhang, Íñigo Goiri, Brijesh Warrier, Nithish Mahalingam, and Ricardo Bianchini. POLCA: Power Oversubscription in LLM Cloud Providers. *arXiv preprint arXiv:2308.12908*, 2023.

[34] David A. Patterson, Joseph Gonzalez, Quoc V. Le, Chen Liang, Lluis-Miquel Munguia, Daniel Rothchild, David R. So, Maud Texier, and Jeff Dean. Carbon Emissions and Large Neural Network Training. In *arXiv Computing Research Repository (CoRR)*, 2021.

[35] PyTorch. Imagenet training in PyTorch. https://github.com/pytorch/examples/tree/main/imagenet.

[36] Mohammad Sadrosadati, Seyed Borna Ehsani, Hajar Falahati, Rachata Ausavarungnirun, Arash Tavakkol, Mojtaba Abaee, Lois Orosa, Yaohua Wang, Hamid Sarbazi-Azad, and Onur Mutlu. ITAP: Idle-Time-Aware Power Management for GPU Execution Units. *ACM Transactions on Architecture and Code Optimization (TACO)*, 16(1):1–26, 2019.

[37] Mustafa Shihab, Karl Taht, and Myoungsoo Jung. GPUdrive: Reconsidering Storage Accesses for GPU Acceleration. In *Workshop on Architectures and Systems for Big Data*, 2014.

[38] Karen Simonyan and Andrew Zisserman. Very Deep Convolutional Networks for Large-Scale Image Recognition, 2015. URL https://arxiv.org/abs/1409.1556.

[39] Peng Sun, Yonggang Wen, Ruobing Han, Wansen Feng, and Shengen Yan. GradientFlow: Optimizing Network Performance for Large-Scale Distributed DNN Training. *IEEE Transactions on Big Data*, 8 (2):495–507, 2019.

[40] Tramm, John, Romano, Paul, Shriwise, Patrick, Lund, Amanda, Doerfert, Johannes, Steinbrecher, Patrick, Siegel, Andrew, and Ridley, Gavin. Performance Portable Monte Carlo Particle Transport on Intel, NVIDIA, and AMD GPUs. *EPJ Web Conf.*, 302:04010, 2024. doi: 10.1051/epjconf/202430204010. URL https://doi.org/10.1051/epjconf/202430204010.

[41] Farui Wang, Weizhe Zhang, Shichao Lai, Meng Hao, and Zheng Wang. Dynamic GPU Energy Optimization for Machine Learning Training Workloads. *IEEE Transactions on Parallel and Distributed Systems*, 33(11):2943–2954, 2021.

[42] Qiang Wang and Xiaowen Chu. GPGPU Performance Estimation With Core and Memory Frequency Scaling. *IEEE Transactions on Parallel and Distributed Systems*, 31(12):2865–2881, 2020.

[43] Qiang Wang, Xinxin Mei, Hai Liu, Yiu-Wing Leung, Zongpeng Li, and Xiaowen Chu. Energy-Aware Non-Preemptive Task Scheduling With Deadline Constraint in DVFS-Enabled Heterogeneous Clusters. *IEEE Transactions on Parallel and Distributed Systems*, 33(12):4083–4099, 2022.

[44] Carole-Jean Wu, Ramya Raghavendra, Udit Gupta, Bilge Acun, Newsha Ardalani, Kiwan Maeng, Gloria Chang, Fiona Aga, Jinshi Huang, Charles Bai, et al. Sustainable AI: Environmental Implications, Challenges and Opportunities. *Proceedings of Machine Learning and Systems*, 4:795–813, 2022.

[45] Cong Xu, Suparna Bhattacharya, Martin Foltin, Suren Byna, and Paolo Faraboschi. Data-Aware Storage Tiering for Deep Learning. In *2021 IEEE/ACM Sixth International Parallel Data Systems Workshop (PDSW)*, pages 23–28. IEEE, 2021.

[46] Rikiya Yamashita, Mizuho Nishio, Richard Kinh Gian Do, and Kaori Togashi. Convolutional neural networks: an overview and application in radiology. *Insights into imaging*, 9:611–629, 2018.

[47] Jie You, Jae-Won Chung, and Mosharaf Chowdhury. Zeus: Understanding and Optimizing GPU Energy Consumption of DNN Training. In *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)*, pages 119–139, 2023.

[48] Yijia Zhang, Qiang Wang, Zhe Lin, Pengxiang Xu, and Bingqiang Wang. Improving GPU Energy Efficiency through an Application-transparent Frequency Scaling Policy with Performance Assurance.

In *Proceedings of the Nineteenth European Conference on Computer Systems*, EuroSys '24, page 769–785, 2024.

[49] Ziyang Zhang, Yang Zhao, Huan Li, Changyao Lin, and Jie Liu. DVFO: Learning-Based DVFS for Energy-Efficient Edge-Cloud Collaborative Inference. *IEEE Transactions on Mobile Computing*, 2024.

Apêndice A
Trabalho de apoio

A.1 Hardware Profiler

A componente *hardware profiler* pode ser utilizada de forma a encontrar o período mínimo de inutilização para o qual compensa, em termos energéticos, alterar a frequência para o valor mínimo suportado. Para tal, são aplicadas computações pela **GPU** que se seguem de um período de inutilização (que, inicialmente, tem a duração de 10 milissegundos) e, por fim, novamente computações. O *hardware profiler* verifica a energia consumida durante todo esse processo utilizando a frequência máxima e, de seguida, fazendo a alteração da frequência para o valor mínimo apenas durante o intervalo entre as computações. No final, analisa se o consumo energético foi maior utilizando a frequência mínima ou máxima e vai aumentando o intervalo de 5 em 5 milissegundos até que o consumo energético seja menor quando é utilizado a frequência mínima. De notar que o *hardware profiler* executa 10 vezes o processo descrito para cada frequência e intervalo, assim, considera que foi encontrado o período mínimo para o qual compensa alterar a frequência, quando o consumo energético for menor recorrendo à aplicação da frequência mínima durante o período de inutilização em pelo menos 5 das execuções do teste com o intervalo em questão.

Os gráficos descrevem a média dos valores obtidos para cada uma das durações consideradas para o período de inutilização para o ambiente de teste *Commodity* (Figura 43) e *Deucalion* (Figura 44). Como é possível observar, para o *Commodity*, o período mínimo de inutilização para o qual compensa aplicar a frequência mínima é de 60 milissegundos, enquanto que, para o *Deucalion* é de 45 milissegundos.

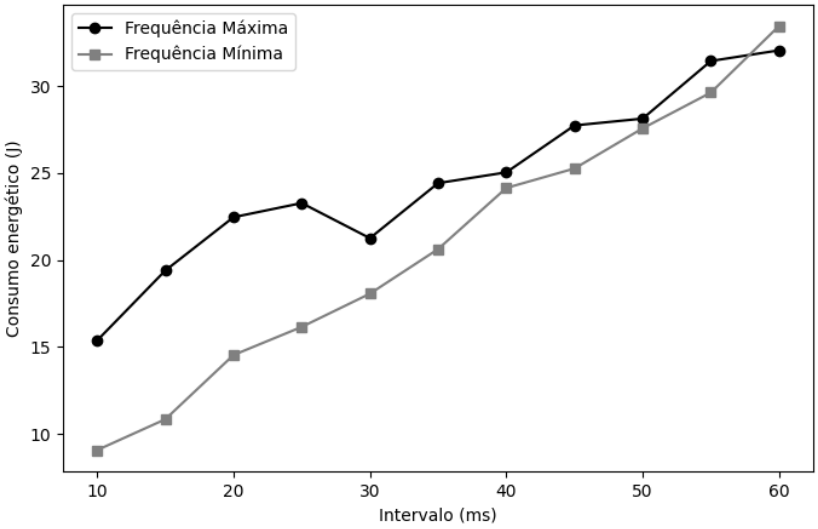


Figura 43: Consumo energético por intervalo de inutilização e frequência aplicada no ambiente de teste *Commodity.*

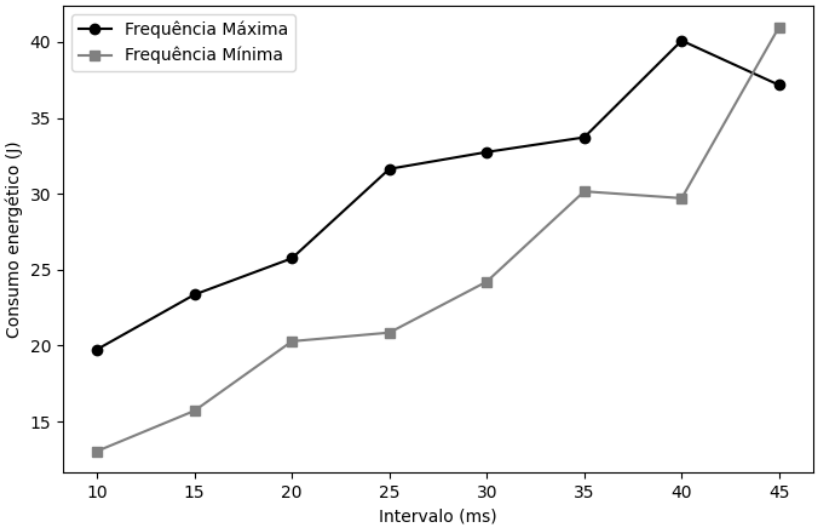


Figura 44: Consumo energético por intervalo de inutilização e frequência aplicada no ambiente de teste *Deucalion.*