NOVA SCHOOL OF SCIENCE & TECHNOLOGY

DEPARTAMENTO DE INFORMÁTICA

FRANCISCO MANUEL ÉVORA ANTÓNIO

Licenciado em Engenharia Informática

ESCALONAMENTO INTELIGENTE DE COMPUTAÇÕES EM AMBIENTES HÍBRIDOS CPU/GPU

MESTRADO EM ENGENHARIA INFORMÁTICA

Universidade NOVA de Lisboa
Setembro, 2021

NOVA SCHOOL OF SCIENCE & TECHNOLOGY

DEPARTAMENTO DE INFORMÁTICA

ESCALONAMENTO INTELIGENTE DE COMPUTAÇÕES EM AMBIENTES HÍBRIDOS CPU/GPU

FRANCISCO MANUEL ÉVORA ANTÓNIO

Licenciado em Engenharia Informática

Orientador: Hervé Miguel Cordeiro Paulino
Professor Associado, Universidade NOVA de Lisboa

Júri:

Presidente: Doutor João Miguel da Costa Magalhães
Professor Associado com Agregação, Faculdade de Ciências e Tecnologia da UNL - Dep. de Informática

Arguente: Doutor Nuno Filipe Valentim Roma
Professor Associado, IST - Dep. Eng. Electrotécnica e de Computadores

Orientador: Doutor Hervé Miguel Cordeiro Paulino
Professor Associado, Faculdade de Ciências e Tecnologia da UNL - Dep. de Informática

MESTRADO EM ENGENHARIA INFORMÁTICA

Universidade NOVA de Lisboa
Setembro, 2021

Escalonamento inteligente de computações em ambientes híbridos CPU/GPU

Agradecimentos

Em primeiro lugar gostaria de agradecer ao meu orientador, o Professor Doutor Hervé Paulino pelo seu apoio incansável no desenvolvimento desta tese. Gostaria também de agradecer à minha família, nomeadamente aos meus pais e aos meus avós. Gostaria ainda de agradecer ao Professor João Lourenço por disponibilizar o *template* [24] de LaTeX com que foi escrita esta tese.

Finalmente, gostaria ainda de agradecer à FCT-UNL por me ter recebido nestes cinco anos.

Resumo

Muitos sistemas computacionais hoje em dia possuem um *host* CPU e uma ou mais GPUs mas nem sempre é fácil tirar partido de todo o potencial destes máquinas. Para podermos ter um grande de aproveitamento do *hardware* é necessário ter um profundo conhecimento da sua arquitetura e efetuar ajustes quando se muda de um sistema para outro. O desempenho de uma execução, tanto na CPU como na GPU, depende de múltiplos fatores, como por exemplo o número de *threads* ou a localidade dos dados. O processo de escalonamento tem de ter em consideração qual é a melhor parametrização para cada processador e qual dos processadores usar.

Esta tese foi desenvolvido no âmbito da *framework* Marrow. Esta *framework* é utilizada para a orquestração de computações em sistemas heterogéneos, permitindo, através de uma computação de alto nível, executar o código na CPU ou na GPU.

O trabalho desenvolvido foi um módulo para a *framework* Marrow que, dado uma certa computação, decide se deve ser executada na CPU ou na GPU. Este módulo encontra também o tamanho do bloco de *threads* que minimiza o tempo de execução, caso o dispositivo escolhido seja a GPU. Para efetuar as decisões foi utilizado um algoritmo de Aprendizagem Automática chamado *Artificial Neural Networks*. São empregues várias redes neuronais e as *features* utilizadas por estas são extraídas a partir das computações Marrow.

Os resultados foram muito positivos, com uma eficácia superior a 97% na escolha do dispositivo onde deve ser executada a computação. Relativamente à previsão do tamanho de bloco ideal, os resultados foram também bastante bons. O tamanho de bloco devolvido pelo módulo resulta num tempo de execução que fica, em média, a menos de 2,4% do tempo de execução mínimo, alcançando valores abaixo de 1% em alguns testes. Foram realizados testes em dois sistemas bastante diferentes e consoante vários aspetos.

Com este módulo, qualquer utilizador da *framework* Marrow pode executar computações com configurações muito próximas das ideais mesmo que não tenha um conhecimento aprofundado da *framework* ou da arquitetura do sistema.

Palavras-chave: Computação heterogénea, *autotuning*, Aprendizagem Automática, parâmetros GPU

Abstract

Nowadays, heterogeneous systems with one CPU host and one or more GPUs are available in many computational systems, but it is not always easy to take full advantage of these machines. To achieve the full potential of the hardware we need to have a profound knowledge of the system architecture and make adjustments when we change from one system to another. The performance of an execution, both in the CPU and the GPU, depends on several factors, like the number of threads or data locality. The scheduling process has to take in consideration which is the best parameterization for each processor and when to assign tasks to each one.

This thesis was developed in the scope of the Marrow framework. This framework is used for the scheduling of computations in heterogeneous systems. From a high level computation it allows you to run the code on a CPU or a GPU.

In this thesis we developed a module for the Marrow framework which, given a certain computation, decides whether to execute it on the CPU or the GPU. This module also finds the thread block size that minimizes execution time, if the chosen device is the GPU. The module uses a Machine Learning algorithm called Artificial Neural Networks to make the decisions. Several neural networks are employed and the features used by these are extracted from the Marrow computations.

The results were very good, with an effectiveness above 97% in choosing the correct device to execute the computation. Regarding the block size, the results were also very good. The block size returned by the module produces an execution time that is, on average, less than 2,4% higher than the minimum execution time, reaching values below 1% in some tests. The testing was performed in two very different systems and evaluated in several different parameters.

With this module, any user of the Marrow framework can execute the computations with configurations very close to optimal, even without a profound knowledge of the framework or the system architecture.

Keywords: Heterogeneous computing, autotuning, Machine Learning, GPU parameters

Índice

Índice de Figuras

Índice de Tabelas

Índice de Listagens

Siglas

ADAM Adaptive Moment Estimation 45
ANN Artificial Neural Network 16, 18, 23, 25, 27, 42, 65

BMU Best Matching Unit 26

CPU Unidade Central de Processamento xiii, xiv, xv, xvii, 1, 2, 3, 4, 7, 9, 10, 11, 12, 13, 15, 16, 17, 27, 28, 36, 37, 38, 46, 47, 48, 51, 52, 54, 55, 56, 58, 59, 62, 64, 65, 66

GPGPU Computação de Carácter Geral em Unidades de Processamento Gráfico 1, 6
GPU Unidade de Processamento Gráfico xii, xiii, xiv, xv, xvii, 1, 2, 3, 4, 6, 7, 8, 9, 10, 11, 12, 13, 15, 16, 17, 27, 28, 29, 34, 35, 36, 37, 38, 40, 41, 43, 45, 46, 47, 48, 51, 52, 54, 56, 57, 58, 59, 60, 61, 62, 63, 64, 65, 66

JIT just in time 17

MLP Multilayer Perceptron xii, 23, 24

PCA Principal Component Analysis 25

SCT Skeleton Computational Tree 10
SM Streaming Multiprocessor xii, 6, 7, 8, 9
SOM Mapa Auto-Organizado 25
SVM Support Vector Machine xii, 16, 17, 22, 23

WRF Weather Research and Forecast 1

1

Introdução

1.1 Motivação

As Unidades de Processamento Gráfico (GPUs) foram criadas com o intuito de acelerar o processamento gráfico nos sistemas computacionais. No entanto, com o aumento do seu poder de computação, este tipo de processadores começou a ser utilizado para executar computações de caráter mais geral. A técnica de utilizar GPUs como processadores adicionais em conjunto com a Unidade Central de Processamento (CPU) para executar computações de caráter geral, ou seja, não gráficas tem o nome de Computação de Carácter Geral em Unidades de Processamento Gráfico (GPGPU) [21]. As computações na CPU e GPU são efetuadas em paralelo o que pode permitir obter melhores desempenhos do que utilizando apenas a CPU. A parte sequencial do programa é habitualmente executada na CPU enquanto que a parte que pode ser paralelizada executa na GPU. A técnica de GPGPU assume uma relevância cada vez maior, uma vez que quase todos os sistemas computacionais com os quais interagimos no dia a dia, quer sejam computadores ou telemóveis, possuem GPUs.

As plataformas base mais utilizadas para execução de computações na GPU são OpenCL [34] e CUDA [10]. Sobre estas foram desenvolvidas muitas outras plataformas para a execução de computações em ambientes heterogéneos CPU/GPU. Entre estes temos o Tensorflow [1]. Tensorflow é uma biblioteca de Aprendizagem Automática e inteligência artificial capaz de executar em várias CPUs e GPUs que utiliza a plataforma CUDA. Outra aplicação é na plataforma Advanced Simulation Library [5]. Esta plataforma de simulação implementada em OpenCL pode ser executada em FPGAs, DSPs ou GPUs. GPGPU pode ainda ser aplicado no ramo das previsões meteorológicas. Uma implementação[32] em CUDA de uma parte do modelo Weather Research and Forecast (WRF) [39] conseguiu alcançar um *speedup* de oito vezes nessa parte do modelo. Esta alteração causou um *speedup* de 1,23 vezes em todo o modelo.

No entanto, nem sempre é fácil tirar o máximo partido de cada máquina. Para haver o máximo aproveitamento do *hardware* é necessário que o programador tenha um conhecimento aprofundado do mesmo e possua experiência em programação em OpenCL ou

CUDA. Para além disso, é necessário adaptar o código sempre que exista uma mudança a nível de *hardware*.

Por estas razões faz sentido ter abstrações de alto nível unificadoras, que escondam do programador os detalhes intrínsecos à programação de cada um destes tipos de processadores. Através da especificação de uma computação em alto nível deve ser encontrado o melhor dispositivo para executar a computação e o tamanho de bloco para a GPU que minimiza o tempo de execução. Isto seria ideal porque permitiria a qualquer pessoa tirar partido de todo o poder de computação de um sistema, mesmo que não tenha um profundo conhecimento da arquitetura do sistema atual. Por outro lado, a execução destas computações no sistema CPU/GPU tem de ser eficiente, caso contrário não faz sentido investir recursos no processo de paralelização do código.

1.2 Problema

O escalonamento de computações em sistemas compostos por um *host* CPU e uma ou mais GPUs constitui um problema cuja resolução não é nada trivial. Isto acontece porque o escalonamento ótimo está dependente de uma grande quantidade de fatores que dificilmente podem ser considerados em tempo útil.

As arquiteturas dos sistemas podem ser extremamente diferentes o que faz com que essas arquiteturas sejam melhores para executar diferentes tipos de computações. Consoante a arquitetura e o estado do sistema pode ser mais proveitoso efetuar uma certa computação na CPU ou na GPU. Para podermos tomar uma decisão informada é necessário termos um descritivo da aplicação.

Para serem lançadas aplicações na GPU é necessária uma certa orquestração que tem o seu *overhead*. Para além disso, a transferência dos dados para a GPU também tem um *overhead*. Em certos casos todo este *overhead* pode fazer com que seja mais rápido executar toda a computação na CPU. Noutros casos a computação pode ser tão extensa e complexa que seja benéfico executar essa computação na GPU.

O *input* pode também ter uma grande influência no escalonamento ótimo de uma computação. Quando o tamanho do *input* a ser processado é maior pode ser melhor executar a computação na GPU, enquanto que nos casos de um *input* de menores dimensões pode ser melhor efetuar essa computação na CPU. Nos casos em que o *input* tem influência no escalonamento das computações as decisões sobre a execução só podem feitas em tempo de execução. No entanto, nos casos em casos em que o *input* não tem influência no escalonamento todas as decisões podem ser feitas em tempo de compilação.

Outro aspeto a ter em consideração é a utilização de vários *kernels* sucessivamente. Isto é importante porque se um programa utiliza vários *kernels* consecutivamente podemos amortizar o custo das transferências entre CPU e GPU uma vez que não é feita a transferência apenas devido a um único *kernel*.

A execução na CPU e GPU em si também pode ser ajustada através de diferentes parâmetros. Na CPU podemos ajustar o número de *threads* a utilizar. Na GPU existem mais

parâmetros ajustáveis como o número de *threads* por cada bloco ou a memória partilhada por bloco, entre outros. Estes parâmetros, que podem ser decididos tanto em tempo de compilação como em tempo de execução, têm uma grande influência na performance do programa. Contudo, a exploração de todas as configurações é um processo bastante extenso e demorado, pelo que são necessários algoritmos que facilitem esta procura.

Por último, o estado do sistema também pode afetar o escalonamento ótimo. A carga da CPU, por exemplo, pode afetar este escalonamento fazendo com que o programa seja executado em menos tempo se alguma da computação for repartida pela GPU.

Todos estes fatores tornam a decisão sobre onde e como executar a computação extremamente difícil e demorada para ser efetuada manualmente em tempo de execução. Existem várias abordagens para resolver este problema como a pesquisa iterativa das melhores configurações ou o uso de heurísticas mas a que estamos mais interessados é o uso de Aprendizagem Automática. Aprendizagem Automática é o ramo da inteligência artificial que se foca no reconhecimento de padrões, o que pode ser bastante útil para efetuar as decisões sobre o escalonamento das computações. Já existem alguns trabalhos que utilizam Aprendizagem Automática para efetuar o *autotuning* de um programa [12, 27, 13, 7, 33, 42, 48, 25, 46, 45, 17, 2, 14, 44, 19, 11, 6], enquanto que outros trabalhos utilizam abordagens diferentes para efetuar o *autotuning* [16, 26, 15, 18, 40, 28, 49, 47]. Todos estes trabalhos serão detalhados na secção 3.1.

O fator diferenciador deste trabalho é que procura otimizar o lançamento do *backend* em tempo de compilação. Tanto quanto sabemos, mais nenhum trabalho efetua esta otimização neste momento. Outro aspeto neste trabalho que não é comum é que funciona de forma estática e dinâmica.

1.3 Solução

Nesta tese vamos utilizar uma *framework* chamada Marrow [29]. Marrow é uma *framework* para a orquestração de computações OpenCL executadas na GPU, detalhada na secção 2.2. Atualmente o Marrow possui um *backend* OpenCL para GPU e um *backend* C++ para CPU.

O trabalho desenvolvido nesta tese consiste em acrescentar um módulo à *framework* Marrow que, dado uma certa computação, decida se essa computação deve ser executada na CPU ou na GPU. Este módulo permite ainda encontrar o tamanho de bloco ideal para a GPU. Existem várias abordagens para alcançar isto como o uso de heurísticas ou a pesquisa iterativa das melhores configurações mas a abordagem em que nos vamos focar é o uso de Aprendizagem Automática.

1.4 Contribuições

As principais contribuições desta tese são:

- Um sistema capaz de criar modelos de Aprendizagem Automática específicos para a máquina alvo da execução das computações.
- A partir de *features* extraídas de uma computação Marrow, estes modelos serão capazes de prever qual o melhor *backend* disponível para executar essa mesma computação.
- Caso esse *backend* seja o GPU, os modelos prevêem o tamanho do bloco de *threads* que minimiza o tempo de execução.
- Incorporação deste sistema na *framework* Marrow sob a forma de um módulo.
- Avaliação do trabalho desenvolvido relativamente à escolha do *backend* e do tamanho do bloco de *threads*. Avaliámos também o tempo de resposta da aplicação e a duração do processo de criação do *dataset*. Apesar de este não ser o principal foco desta tese, analisámos também os valores previstos para o tempo de execução nos *backends* testados, CPU e GPU. Todas estas avaliações foram feitas consoante vários parâmetros e em dois sistemas distintos.

1.5 Estrutura do Documento

Esta tese encontra-se dividida em seis capítulos, cada um com várias secções.

Neste capítulo apresentamos a motivação para esta tese, o problema que é tentado resolver e em que consiste a nossa solução.

No capítulo 2 introduzimos a *framework* Marrow que vamos utilizar nesta tese. Apresentamos um exemplo de uma árvore criada pelo Marrow e falamos da arquitetura desta *framework*. Além disso, explicamos também o modelo de execução das GPUs.

No capítulo 3 falamos dos trabalhos de *autotuning* já existentes, quer utilizem Aprendizagem Automática ou não. Temos uma tabela que resume todos os trabalhos de *autotuning* e uma tabela com os trabalhos mais parecidos com o nosso que utilizam Aprendizagem Automática. Temos também uma secção onde falamos de trabalhos que utilizam Aprendizagem Automática na compilação de programas, especificamente para ajudar o compilador a efetuar as melhores decisões. Neste capítulo falamos também de Aprendizagem Automática supervisionada e não supervisionada e enumeramos as técnicas mais utilizadas.

No capítulo 4 apresentamos a solução implementada. Começamos por mostrar de que maneira o trabalho desenvolvido se enquadra na *framework* Marrow e uma visão geral desse mesmo trabalho. Em seguida, explicamos as opções tomadas na construção do *dataset* e na construção das redes neuronais. Esclarecemos ainda o processo de escolha do *backend* e a utilização de caches.

No capítulo 5 procedemos à avaliação da solução desenvolvida. São apresentados os objetivos e os resultados obtidos, assim como a metodologia de avaliação e o ambiente de testes utilizado.

Por último, no capítulo 6 são expostas as conclusões desta dissertação e é discutido o trabalho que pode ser desenvolvido no futuro.

2

Marrow e GPUs

Neste capítulo vamos falar sobre o modelo de programação e execução das Unidades de Processamento Gráfico (GPUs). Vamos também falar sobre a *framework* que vamos usar nesta tese chamada Marrow, explicando o seu modelo de programação, a sua arquitetura e o seu modelo de execução.

2.1 Computação de Carácter Geral em GPUs

Nesta secção vamos explicar um pouco da arquitetura das GPUs e da forma como podem ser utilizadas para a execução de computações de caráter geral (Computação de Carácter Geral em Unidades de Processamento Gráfico (GPGPU)).

2.1.1 Arquitetura das GPUs

Agora vamos falar um pouco da arquitetura das GPUs. Na figura 2.1 podemos observar a estrutura de uma GPU NVIDIA Volta GV100.

A arquitetura de uma GPU é formada por vários SMs compostos por bastantes *cores* que podem executar em paralelo. Cada um destes *cores* executa o código OpenCL ou CUDA, que será explicado em maior detalhe a seguir.

Na figura 2.2 podemos ver o esquema de um SM e na figura 2.3 temos um esquema com os diferentes tipos de memória da GPU. Como podemos verificar, cada SM contém três tipos de memória. A memória que apresenta menor tempo de acesso são os registos, seguidos da memória partilhada. Para além disso, cada SM possui um conjunto de caches que contém dados transferidos anteriormente da memória do dispositivo GPU. Os acessos a memória do SM são sempre mais rápidos que os acessos a memória da GPU. Cada SM têm também vários componentes importantes chamados *warp schedulers*. Um *warp scheduler* é a unidade responsável pelo escalonamento de computações para cada *core* do SM. Na figura 2.3 podemos também verificar que existem vários tipos de memória no dispositivo GPU. A memória global e a memória local são os tipos de memória que apresentam os maiores tempos de acesso. A memória global é comum a todos os SMs do dispositivo e a memória local contém variáveis específicas a cada SM e que não foram

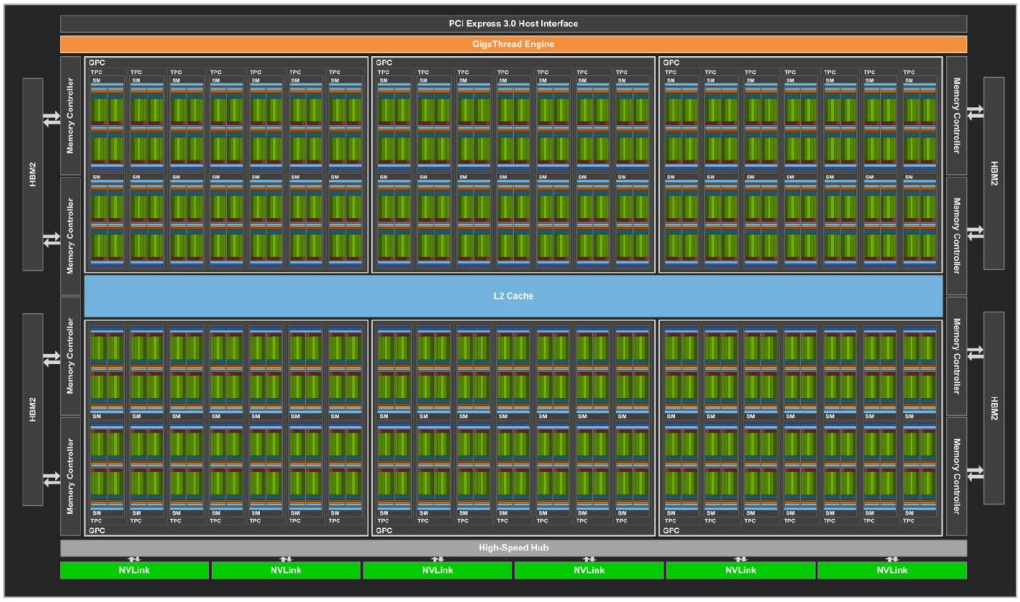


Figura 2.1: Esquema da GPU NVIDIA Volta GV100 com oitenta e quatro Streaming Multiprocessors (SMs) [37].

colocadas nos registos por falta de espaço. A memória constante contém valores que nunca são alterados e são colocados na cache do SM após consulta. A memória de texturas contém valores apenas de leitura e possui um tempo de acesso constante. Os valores desta memória são também colocados na cache do SM após consulta.

2.1.2 Execução nas GPUs

Para poder executar uma computação na GPU é necessário escrever código específico para a GPU. Este código, referido como *kernel*, pode ser escrito em OpenCL ou CUDA. Por exemplo, para executar o seguinte ciclo que soma dois vetores:

```
for(int i = 0; i < size; i++)
    a[i] = b[i] + c[i];
```

É necessário escrever os respetivos *kernels*. Na listagem 1 temos o *kernel* em OpenCL e na listagem 2 temos o *kernel* em CUDA. O *kernel* é o código executado por cada *thread* da GPU. As *threads* da GPU são diferentes das *threads* da Unidade Central de Processamento (CPU) na medida em que são mais simples e o número de *threads* da GPU é muito superior ao da CPU. Como podemos observar nas listagens, em ambos os *kernels* é necessário obter o *id* da *thread* para saber qual a posição dos vetores que deve ser lida e escrita. É verificado se o *id* da *thread* é inferior ao tamanho dos vetores (*size*) para ter a certeza que não são acedidas posições fora dos limites. Para além do código que será executado na GPU é também necessário o código que será executado no *host* CPU. Neste caso, o *host* necessita de transferir os vetores *b* e *c* para a GPU antes da execução do *kernel* e necessita de transferir o vetor *a* para o *host* quando a execução for concluída. Em relação à

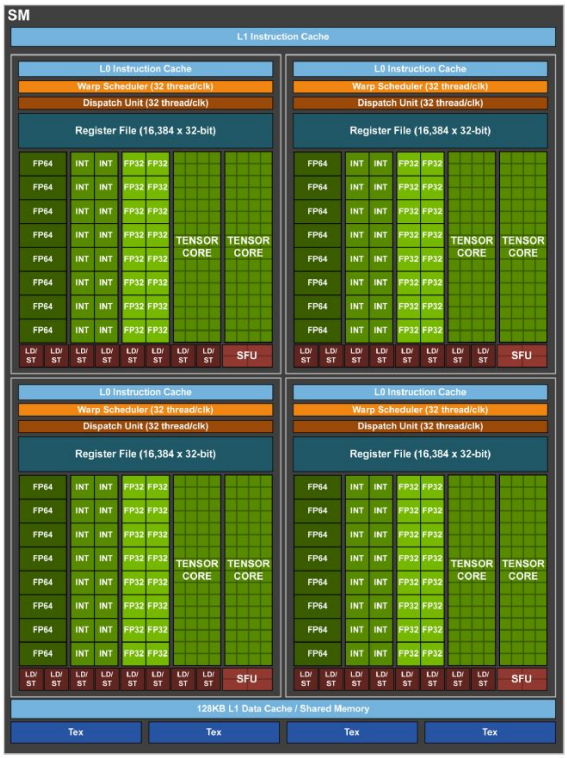


Figura 2.2: Esquema de um SM da GPU NVIDIA Volta GV100 [37].

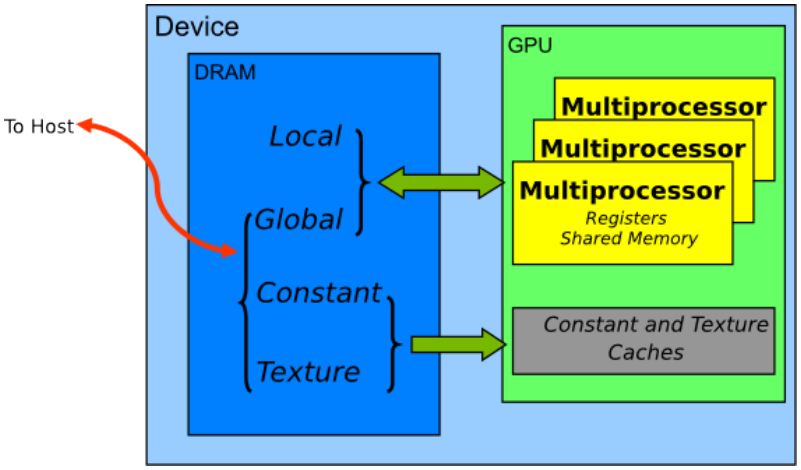


Figura 2.3: Esquema da memória de uma GPU [36].

especificação do número de *threads* com que deve ser executado o *kernel*, existem algumas diferenças entre CUDA e OpenCL.

Em CUDA é especificado o tamanho do bloco de *threads* (block_size) e o tamanho da grelha (grid_size). O tamanho da grelha corresponde ao número de blocos de *threads*. Para lançar o *kernel* na listagem 2 utilizamos:

```
vector_add<<<grid_size, block_size>>>(a, b, c, size);
```

Em que os vetores a, b e c foram alocados no dispositivo GPU e *size* é o tamanho destes

Listing 1 *Kernel* OpenCL que soma dois vetores.

```
__kernel void vector_add (__global float* a, __global float* b,  __global float* c,
                          int size){
    const int id = get_global_id (0);
    if(id < size)
        a[id] = b[id] + c[id];
}
```

Listing 2 *Kernel* CUDA que soma dois vetores.

```
__global__ void vector_add(double *a, double *b, double *c, int size){
    int id = blockDim.x * blockIdx.x + threadIdx.x;
    if(id < size)
        a[id] = b[id] + c[id];
}
```

mesmos vetores.

Em OpenCL o *kernel* é executado da seguinte maneira:

```
clStatus = clEnqueueNDRangeKernel(command_queue, kernel, 1, NULL,
                          &global_size, &local_size, 0, NULL, NULL);
```

O objeto *command_queue* é a fila de comandos onde será colocado o *kernel* para ser executado e o objeto *kernel* contém o código na listagem 1 e os respetivos argumentos. O valor de *global_size* é o número de *work items* que devem ser executados, ou seja, é o tamanho dos vetores. O valor de *local_size* é número de *work items* que devem ser colocados em cada *work group*.

Para executar uma computação são atribuídos blocos de *threads* aos SMs . O tamanho de cada bloco é programável, as *threads* dentro do mesmo bloco partilham dados e são sincronizadas. As *threads* de cada bloco são agrupadas em conjuntos de trinta e duas *threads* chamados *warps*. Cada *warp* funciona em modo *lock-step* o que significa que todos os *warps* em cada bloco executam o mesmo código GPU mas com dados diferentes. Cada *thread* é executada por um *core* da GPU. Todos os *cores* do mesmo SM executam a mesma computação mas com dados diferentes. Cada GPU executa uma grelha de blocos que pode ter uma, duas ou três dimensões, tal como os blocos de *threads*. Dentro de uma grelha todas as *threads* executam o mesmo *kernel*.

Uma das propriedades da execução de uma computação numa GPU é a necessidade de transferir os dados e o código do *host* CPU para a GPU antes da execução da computação e no sentido contrário no fim da execução. Isto causa um certo *overhead* que é necessário ter em consideração quando é decidido o escalonamento.

2.2 Marrow

Marrow [29, 3] é uma *framework* para C++ com intuito de facilitar a orquestração de computações paralelas em CPU e GPU. Atualmente possui um *wrapper* OpenCL [34] para GPU e estão em desenvolvimento backends para CUDA (para GPU) e OpenMP [9] (para CPU). Esta *framework* oferece uma série de *skeletons* paralelos que podem ser combinados para obter comportamentos específicos. Desta forma, uma computação Marrow define uma árvore de *skeletons* em que cada cada *skeleton* aplica um comportamento específico à sua sub-árvore, até às folhas que são computações OpenCL ou OpenMP. Marrow é responsável pela orquestração necessária para executar estas árvores em ambientes multi-CPU e multi-GPU, incluindo a ordenação dos pedidos de transferência de dados e execução, mas também a comunicação entre nós da árvore.

2.2.1 Modelo de programação

Para explicar o modelo de programação do Marrow vamos começar com um exemplo. Esta é a expressão utilizada como exemplo:

$$\sqrt{\sum_i (a_i - b_i)^2}$$

em que a e b são dois *containers* (*arrays* ou vetores). A sua implementação em Marrow pode ser dada pela simples expressão:

```
double s = sqrt(reduce<plus>(pow(a - b)));
```

que dá origem à árvore representada na figura 2.4. A estas árvores, geradas pelo Marrow, damos o nome de Skeleton Computational Trees (SCTs). Estas são árvores de *skeletons* em que cada *skeleton* aplica um comportamento específico à sua sub-árvore.

Decompondo a expressão nos seus vários passos, para uma mais fácil explicação e compreensão, temos:

```
auto c = a - b;  // ai - bi
auto d = pow(c, 2);   // (ai - bi)^2
auto soma = reduce<plus>(d);
double s = sqrt(soma);
```

Como podemos verificar, utilizámos *auto* em vez de um tipo de *container*. Ao utilizarmos a *keyword auto* estamos a indicar que o tipo da variável deve ser inferido pelo compilador. Se atribuirmos a expressão a um tipo de *container* a sua computação é imediatamente desencadeada. Desta forma, o uso de *auto* permite-nos ir construindo a árvore sem nunca a executar.

Para a GPU são criados dois *kernels*, um executa a operação `pow(a-b, 2)` e outro faz a redução porque esta necessita de uma barreira, isto é, a redução só pode começar quando

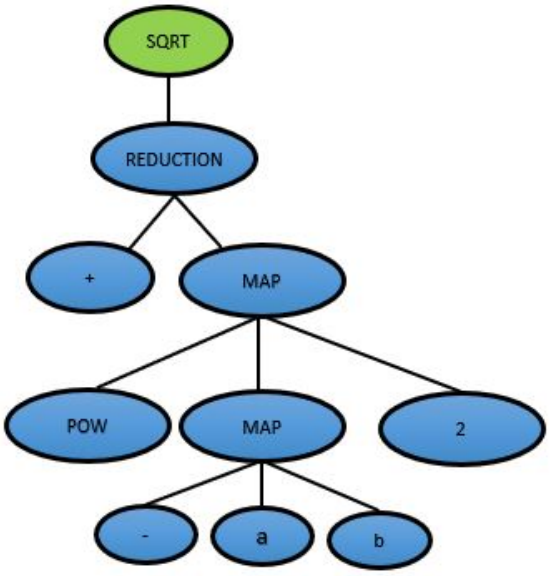


Figura 2.4: Árvore gerada pela *framework* Marrow.

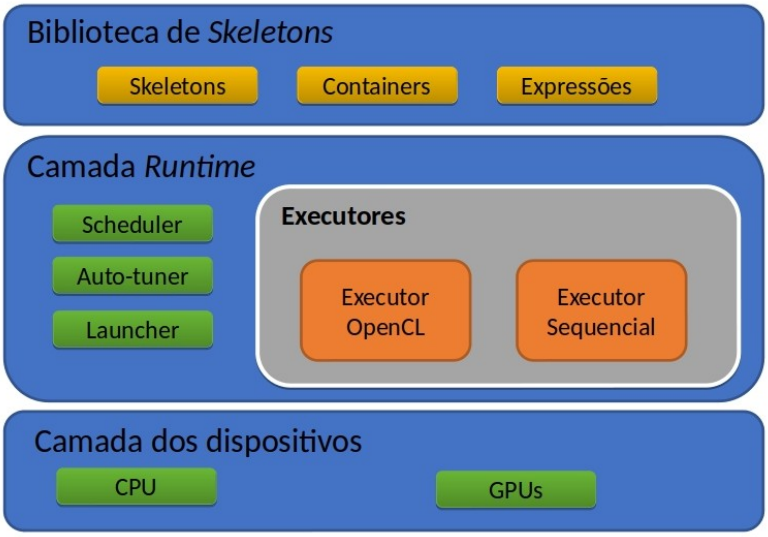


Figura 2.5: Arquitetura da *framework* Marrow.

todos os valores do vetor já tiverem sido calculados. Por isso, a redução só pode ser feita depois de ser desenvolvido o controlo ao *host*. A raiz quadrada é a única operação feita na CPU, todas as outras são feitas na GPU. A partir desta árvore podemos obter as *features* que serão utilizadas no modelo de Aprendizagem Automática que será falado no capítulo 3. Uma das *features* que pode ser extraída da árvore são os acessos a memória.

2.2.2 Arquitetura e modelo de execução

Na figura 2.5 podemos ver um esquema da arquitetura da *framework* Marrow. A arquitetura é composta por três camadas distintas, a camada da biblioteca de *skeletons*, a camada

runtime e a camada dos *backends*, com *backends* para OpenCL, OpenMP (em desenvolvimento) e um *backend* sequencial.

Na camada da biblioteca de *skeletons* estão incluídos os *skeletons*, os *containers* e as expressões. O Marrow contém vários *skeletons* como o *map*, *filter*, *reduce* e *scan*. Os *containers* existentes são *arrays*, vetores e tensores. As expressões permitem efetuar operações, chamar funções, assim como atribuir valores e aceder a variáveis sobre os *containers*. Estas operações são depois passadas à camada do *runtime* para que possam ser executadas pelo executor configurado.

No *runtime* do Marrow está incluído o executor de OpenCL e o executor sequencial, como podemos ver na figura 2.5. O executor OpenCL gera o *kernel* num ficheiro à parte, caso este já exista vai buscá-lo e caso já esteja compilado executa-o, se o *kernel* não existir o executor cria-o e compila-o. Este executor gera ainda o código do *host*. O executor sequencial gera simplesmente o código a executar. O *auto-tuner* é um componente responsável por manter equilibradas as computações na CPU e GPU.

Na camada mais abaixo temos os dispositivos CPU e GPU onde as computações podem executar.

Relativamente ao modelo de execução, a *framework* Marrow gera uma árvore quando pretendemos executar uma computação. Estas árvores são submetidas ao *runtime* e o *launcher* do *runtime* chama o executor para proceder à sua execução. A partir da árvore são gerados os *kernels*. O executor chama o *scheduler* que executa estes *kernels* numa lógica de *first come first served*, garantindo que apenas os *kernels* que têm as suas dependências de dados resolvidas estão prontos a executar. No caso do OpenMP é gerado código OpenMP para correr no *host* CPU. No caso do OpenCL é gerado código OpenCL para correr na GPU e o código para o *host*.

3

Autotuning e Aprendizagem Automática

Neste capítulo vamos falar sobre os trabalhos já existentes relativos a *autotuning* de computações na Unidade Central de Processamento (CPU) e Unidade de Processamento Gráfico (GPU), com e sem Aprendizagem Automática. Vamos também falar sobre a utilização de Aprendizagem Automática na compilação de programas e explicar algumas das técnicas de Aprendizagem Automática mais comuns.

3.1 *Autotuning*

Nesta secção iremos apresentar os trabalhos já existentes relacionados com *autotuning* de computações na CPU e GPU. A tabela 3.1 contém um resumo desses trabalhos. Foi feita uma pesquisa sobre estes trabalhos com especial atenção ao seguinte conjunto de características.

Arquitetura alvo: Um dos aspetos que nos interessa nestes trabalhos é a arquitetura alvo. Esta é a arquitetura do sistema no qual é feito o *autotuning*. Este sistema pode ser composto apenas por uma ou mais CPUs, como nos trabalhos [33, 42, 2, 13, 45, 14], apenas por uma ou mais GPUs, como nos trabalhos [48, 16, 27, 18, 44] ou por uma ou mais CPUs e uma ou mais GPUs, como nos trabalhos [25, 11, 7, 26, 17, 15, 46, 19, 12, 28, 49]. No nosso caso o sistema terá uma arquitetura alvo CPU/GPU.

Linguagem origem/linguagem alvo: Outros aspetos que nos interessam são a linguagem fonte e a linguagem alvo. A linguagem origem é a linguagem em que os programas são escritos, ou seja, a linguagem do programa antes de ser efetuado o *autotuning*. Por outro lado, a linguagem alvo é a linguagem resultante do *autotuning*. Na maior parte dos trabalhos encontrados as linguagens alvo e origem são as mesmas. As linguagens alvo mais utilizadas são OpenCL [11, 16, 17, 15, 27, 46, 18, 40, 12, 49] e CUDA [48, 7, 16, 26, 18, 6]. No nosso caso a linguagem origem é C++ e as linguagens alvo são OpenCL ou CUDA.

Paper	Alvo	Considera a carga da CPU	Considera a localização/ volume dos dados	Objetivo	Modo de *autotuning*	Linguagem origem	Linguagem alvo	Abordagem	Estático/ dinâmico	*Offline / online learning*
[48]	GPU	Não	Não	Performance	Otimização de programas para GPU consoante o *input*	CUDA	CUDA	AA	Dinâmico	*Offline*
[45]	CPU	Não	Sim	Performance	Otimização do número de *threads*	OpenMP	OpenMP	AA	Estático	*Offline*
[25]	CPU/GPU	Não	Não	Performance	Distribuição de ciclos pela CPU e GPU	C/C++	Código máquina	AA	Dinâmico	*Online*
[11]	CPU/GPU	Não	Sim	Performance	Otimizações das transferências de dados e configuração da GPU	C++	OpenCL / OpenMP	AA	Dinâmico	*Offline*
[7]	CPU/GPU	Não	Não	Performance	Otimização de código	CUDA	CUDA	AA	Estático	*Offline*
[16]	GPU	Não	Não	Performance	Otimizações de ciclos	C + HMPP	OpenCL / CUDA	Minimização do tempo de execução	Estático	*Offline*
[26]	CPU/GPU	Sim	Não	Performance e consumo de energia	Distribuição das computações pela CPU e GPU de forma igual	CUDA	CUDA	Heurística	Dinâmico	*Online*
[17]	CPU/GPU	Não	Não	Performance	Otimização de ciclos	OpenMP	OpenCL / OpenMP	AA	Dinâmico	*Offline*
[15]	CPU/GPU	Não	Sim	Performance	Otimizada a multiplicação de matrizes	OpenCL	OpenCL	Minimização do tempo de execução	Estático	*Offline*
[27]	GPU	Não	Não	Performance	Otimização do número de *threads*	OpenCL	OpenCL	AA	Estático	*Offline*
[46]	CPU/GPU	Não	Sim	Performance	Otimizado o escalonamento de computações OpenCL	OpenCL	OpenCL	AA	Dinâmico	*Offline*
[18]	GPU	Não	Não	Performance e consumo energético	Otimização dos parâmetros de *kernels* e da GPU	OpenCL/ CUDA	OpenCL/ CUDA	Minimização do tempo de execução	Estático	*Offline*
[40]	CPU/GPU	Sim	Não	Performance	Distribuição das computações por CPU(s) e GPU(s)	OpenCL	OpenCL	Previsão do tempo de execução nos dispositivos	Estático	*Online*
[19]	CPU/GPU	Não	Sim	Performance	Otimizações nas transferências de dados e escolha do dispositivo	Java	PTX	AA	Dinâmico	*Offline*
[12]	CPU/GPU	Não	Não	Performance	Otimização das configurações de OpenCL	OpenCL	OpenCL	AA	Estático	*Offline*
[14]	CPU	Não	N/A	Performance	Otimização das *flags* do compilador	C++	C++	AA	Estático	*Offline*
[28]	CPU/GPU	Não	Não	Consumo de energia	Ajuste dinâmico do *hardware*	N/A	N/A	*Model Predictive Control*	Dinâmico	*Online*
[44]	GPU	Não	Não	Performance	Otimização de operações com matrizes	PTX	PTX	AA	Dinâmico	*Offline*
[47]	GPU	Não	Não	Performance	Procura da melhor configuração do *kernel*	OpenCL/ CUDA/ C	OpenCL/ CUDA/ C	Pesquisa da melhor configuração ou algoritmo genético	Estático	*Offline*
[6]	GPU	Não	Não	Performance	Otimização das configurações da GPU	CUDA	CUDA	AA	Estático	*Offline*
[49]	CPU/GPU	Não	Não	Performance	Otimização do *workgroup* em OpenCL	OpenCL	OpenCL	Minimização do tempo de execução	Estático	*Offline*
Nosso trabalho	CPU/GPU	Não	Sim	Performance	Escolha do dispositivo e das configurações da GPU	OpenCL	OpenCL	AA	Estático e Dinâmico	*Offline*

Tabela 3.1: Trabalhos relacionados com *autotuning*

Abordagem: Para efetuar o *autotuning* podem ser utilizadas várias abordagens. A abordagem em que nos vamos focar mais é a utilização de Aprendizagem Automática [33, 42, 2, 13, 48, 45, 25, 11, 7, 17, 27, 46, 19, 12, 14, 44, 6], mas também existem outras abordagens, como a utilização de heurísticas [26]. Os trabalhos que utilizam Aprendizagem Automática e que mais se assemelham ao nosso são detalhados na secção 3.1.1. No caso da utilização de heurísticas estas são utilizadas para tentar prever qual a melhor distribuição da computação ou quais as melhores configurações para a CPU e GPU. Outra abordagem bastante utilizada é a pesquisa iterativa das melhores configurações. O espaço de pesquisa é reduzido através de heurísticas ou filtragens sucessivas, as alternativas resultantes são executadas e é escolhida a que produz melhores resultados. No nosso caso a abordagem utilizada será Aprendizagem Automática.

Objetivo: O objetivo do trabalho foi outro aspeto que tomámos em consideração. A grande maioria dos trabalhos tem como objetivo aumentar a performance dos programas reduzindo o tempo de execução [33, 42, 2, 13, 48, 45, 25, 11, 7, 17, 27, 46, 19, 12, 14, 44, 6, 16, 15, 40, 49], mas uma pequena parte tem como alternativa [26, 18] ou único objetivo a redução do consumo energético [28]. O objetivo do nosso trabalho será também aumentar a performance reduzindo o tempo de execução.

Considera a carga da CPU: Outro aspeto que tomámos em conta é se no trabalho consideram a carga da CPU quando efetuam a decisão. Isto pode ser importante porque caso a carga da CPU seja alta pode ser mais benéfico distribuir algumas das computações da CPU pela GPU. Poucos trabalhos tiveram isto em consideração[26, 40]. Estes trabalhos procuram equilibrar o tempo de execução entre CPU e GPU de maneira a que se evite que um dos dispositivos fique à espera do outro. Desta forma, a carga da CPU é tida em consideração porque esta se fará notar no tempo de execução.

Estático/dinâmico: Um aspeto muito importante consiste em que momento o algoritmo faz as decisões sobre a execução do programa, isto é, em que momento decide onde deve ser executada a computação e as configurações necessárias. Consideramos o algoritmo estático se efetua estas decisões em tempo de compilação e dinâmico se só as efetua em tempo de execução. A abordagem estática [33, 42, 2, 13, 45, 7, 16, 15, 27, 18, 40, 12, 14, 6, 49] tem a vantagem de não causar *overhead* na execução, no entanto a abordagem dinâmica [48, 25, 11, 26, 17, 46, 19, 28, 44] permite adaptar a parâmetros do *runtime*, como por exemplo o *input*.

Considera a localização/volume dos dados: A localização e volume dos dados é também importante quando efetuamos as decisões sobre o escalonamento. Por vezes, o *overhead* causado pela transferência dos dados para a GPU pode fazer com que seja mais rápido executar todas as computações na CPU. Noutras situações, consoante o tamanho do *input*

pode ser mais proveitoso executar a computação na CPU ou na GPU. Poucos trabalhos tiveram este pormenor em conta [45, 11, 15, 46, 19].

***Offline/online learning*:** Outro detalhe dos trabalhos que prestámos atenção está relacionado com o modo de aprendizagem do algoritmo. Consideramos *offline learning* se um programa aprende o modelo uma vez e não volta a fazer alterações a este modelo. *Online learning* é o caso em que o programa vai aprendendo com as novas compilações e execuções fazendo alterações no modelo.

Modo de *autotuning*: Por último, vamos também considerar a área de *autotuning* em que cada trabalho se foca. Alguns trabalhos focam-se na otimização de ciclos [33, 42, 25, 16, 17], outros focam-se em encontrar as melhores configurações da GPU ou CPU [11, 18, 12, 28, 6, 49, 45] e outros focam-se no escalonamento de computações [26, 46, 40], entre outras áreas.

3.1.1 *Autotuning* com Aprendizagem Automática

Nesta secção iremos falar sobre trabalhos referentes a *autotuning* que utilizam Aprendizagem Automática. Um dos aspetos que nos interessa é a técnica de Aprendizagem Automática utilizada. As técnicas mais utilizadas são Aprendizagem Automática supervisionada, especificamente Artificial Neural Network (ANN) e Support Vector Machine (SVM). Outro aspeto importante é a maneira como são extraídas as *features*. As *features* podem ser extraídas estaticamente do código ou de representações intermédias, mas também dinamicamente através de parâmetros do *runtime* como o tempo de execução. Na tabela 3.2 podemos observar os trabalhos de *autotuning* que utilizam Aprendizagem Automática.

Y. Liu *et al.* [48] procuraram encontrar os melhores parâmetros para compilar programas em CUDA dado um certo *input*. Concluíram que o *input* pode ter uma grande influência na otimização do programa e criaram uma *framework* para encontrar automaticamente as melhores otimizações para aplicações GPU. São utilizadas árvores de decisão para criar modelos que efetuam as otimizações consoante o *input* das aplicações. Em tempo de compilação o compilador produz um executável para cada *parameter vector* que é considerado o melhor para alguns *inputs* de treino na *performance database*. Quando o programa é executado com um *input* arbitrário o *wrapper* usa as *regression trees* criadas para determinar rapidamente o executável correto com base no input e corre o programa.

Chi-Keung Luk *et al.* [25] criaram o compilador Qilin que utiliza algoritmos *curve fitting* para estimar o tempo de execução de um programa na CPU e na GPU, dado um certo *input*. O compilador depois usa essa informação para determinar a melhor distribuição de um ciclo pela CPU e GPU. Qilin guarda a previsão do tempo de execução de um programa numa base de dados e na próxima vez que o programa for executado com um *input* diferente esse valor é utilizado para determinar o melhor mapeamento. Utiliza

Aprendizagem Automática supervisionada e a técnica é regressão linear. Qilin disponibiliza uma API para C/C++ e usa compilação dinâmica para converter as chamadas à sua API em código máquina. Para reduzir o *overhead* em tempo de execução maior parte da compilação dinâmica é feita em mode *lazy*. Isto significa que a maior parte dos passos da compilação só é feita quando os resultados da compilação forem necessários.

U. Dastgeer *et al.* [11] criaram uma biblioteca para C++ chamada SkePU com o objetivo de tornar programação em paralelo mais fácil. SkePU utiliza *backends* para GPU em CUDA e OpenCL mas também tem *backends* para CPU em OpenMP. Esta biblioteca também faz otimizações ao nível das transferências de dados entre a CPU e a GPU. Foi utilizado um algoritmo genético para encontrar a melhor configuração para cada *backend* e é feita uma previsão do tempo de execução de cada componente em *off-line*. Para calcular estas estimativas é guardado o tempo de cópia de dados entre a CPU e GPU, tempo de execução do *kernel* e o tempo total de execução. SkePU gera um plano de execução com os vários *backends* indicando em que ordem devem ser utilizados os *backends* e com que configuração.

J. Bergstra *et al.* [7] procuraram otimizar código CUDA. A técnica utilizada foi *regression trees* e foi otimizada a performance. As medidas de performance são usadas para treinar um modelo de Aprendizagem Automática construído em *offline*. As *features* são extraídas a partir do *kernel*.

Yuan Wen *et al.* [46] procuraram encontrar o escalonamento ótimo de computações OpenCL em ambientes heterogéneos CPU/GPU. Em tempo de execução determina quais *kernels* utilizam melhor um dispositivo. As *features* são extraídas estaticamente a partir do código, mas também inclui o *speedup* de executar o programa na GPU e algumas *features* obtidas em tempo de execução. É utilizada Aprendizagem Automática supervisionada e a técnica é SVMs. O *predictor* é criado em *offline* utilizando programas de treino.

A. Hayashi *et al.* [19] criaram um compilador just in time (JIT) por cima do compilador JIT da IBM para Java 8. A funcionalidade adicional deste compilador é permitir especificar se a computação deve ser corrida na CPU ou GPU. Esta funcionalidade utiliza SVMs para decidir em tempo de execução em que dispositivo deve ser corrido o programa. Este compilador efetua também otimizações na sua representação intermédia ao nível das transferências de dados entre CPU e GPU. O modelo de Aprendizagem Automática é construído em *offline* e as *features* são extraídas da representação intermédia do compilador.

Thomas L. Fach *et al.* [12] através de Aprendizagem Automática criaram uma solução para tornar o OpenCL mais portável. A sua abordagem consiste, em primeiro lugar, em escolher configurações aleatoriamente para correr o programa e em medir o seu tempo de execução. Com estes dados é treinado o modelo que depois será utilizado para prever o tempo de execução de todas as configurações. As melhores configurações são escolhidas, é medido o tempo de execução de cada uma e é devolvida a melhor configuração. Utilizaram

Paper	Técnica de AA	Extração de *Features*	Modo de *autotuning*
[48]	Árvores de decisão	Extraídas do *input*	Otimização de programas para GPU consoante o *input*
[25]	Regressão Linear	Extraídas do *input*	Distribuição de ciclos pela CPU e GPU
[11]	Algoritmo genético	N/A	Otimizações das transferências de dados e configuração da GPU
[46]	SVM	Extraídas do código, *speedup* estimado e obtidas em tempo de execução	Otimizado o escalonamento de computações OpenCL
[19]	SVM	Extraídas de representações intermédias do compilador	Otimizações nas transferências de dados e escolha do dispositivo
[12]	ANN	N/A	Otimização das configurações de OpenCL
[6]	*k-means* e *regression trees*	Parâmetros de *runtime* e obtidos do compilador	Otimização das configurações da GPU

Tabela 3.2: Trabalhos relacionados com *autotuning* que utilizam Aprendizagem Automática.

Aprendizagem Automática supervisionada e a técnica utilizada foi ANN. Foi otimizada a performance do programa.

T. Balaiah *et al.* [6] criaram uma *framework* para encontrar as melhores configurações para um programa. Esta *framework* é composta por duas fases e as técnicas de Aprendizagem Automática utilizadas são *regression trees* e *k-means clustering*. As *features* são obtidas do compilador, de parâmetros obtidos em tempo de execução como o *input* e de parâmetros relacionados com a utilização de recursos e a eficiência de certos componentes. É encontrada a melhor configuração sem executar o programa.

3.2 Compilação com Aprendizagem Automática

Aprendizagem Automática pode ser aplicada na compilação de programas para encontrar as melhores otimizações que devem ser aplicadas na sua compilação. Estas otimizações podem variar consoante a arquitetura pelo que é necessário adaptar o programa ao sistema em que será executado.

Um aspeto em que Aprendizagem Automática pode ser utilizada é para encontrar as melhores otimizações para o compilador. Mais precisamente, decidir se um ciclo deve ser *unrolled* e o seu *unroll factor* é uma área em que Aprendizagem Automática já foi utilizada

[33, 42]. Outra área em que pode ser utilizada Aprendizagem Automática é na decisão das melhores *flags* do compilador que devem ser utilizadas[2, 14], o que pode ter um grande impacto na performance do programa. Outra utilização possível é para prever características do programa como o tamanho do código, tempo de execução e compilação com o intuito de efetuar otimizações que melhorem estes fatores[13]. Aprendizagem Automática também pode ser utilizada no mapeamento do programa por parte do compilador, assim como na escolha do número de *threads* a utilizar [45]. A escolha de quais políticas de escalonamento utilizar é uma tarefa complexa em que Aprendizagem Automática também pode ser aplicada.

3.3 Técnicas de Aprendizagem Automática

Aprendizagem Automática é um ramo da inteligência artificial que se foca no reconhecimento de padrões. A partir de um conjunto de dados um programa é capaz de reconhecer padrões nesses mesmos dados e efetuar previsões acerca de casos que não estejam nesse conjunto. Aprendizagem Automática pode ser utilizada, por exemplo, para decidir se um *email* deve ser considerado como *spam* ou em reconhecimento facial [4]. Existem três tipos de Aprendizagem Automática, supervisionada, não supervisionada e *reinforcement learning*. O tipo em que nos vamos focar mais é Aprendizagem Automática supervisionada porque é o tipo mais utilizado na área de *autotuning* mas também iremos um pouco de falar de Aprendizagem Automática não supervisionada. Também apresentaremos algumas das técnicas mais utilizadas.

Existem duas categorias de problemas de Aprendizagem Automática, os problemas de classificação e os problemas de regressão. Os problemas de classificação são problemas em que o valor que será previsto é discreto, como por exemplo efetuar o diagnóstico de um paciente com base nos seus sintomas e características ou classificar um email como *spam* ou não. Os problemas de regressão são problemas em que o valor que será previsto é contínuo, como por exemplo, a previsão do preço de um carro usado [4].

3.3.1 Aprendizagem Automática supervisionada

Aprendizagem Automática supervisionada foca-se em prever certos atributos com um conjunto de dados que contém esses atributos, isto é, são fornecidos exemplos já classificados ao algoritmo para que possa prever novos casos. Este conjunto de dados fornecido é chamado conjunto de treino e é o conjunto que nos permite chegar às várias hipóteses. No conjunto de treino $(x^1, y^1), ..., (x^n, y^n)$ assumimos que existe uma função desconhecida $F(X) : X \rightarrow Y$ e o objetivo é encontrar uma função $g(\theta, X) : X \rightarrow Y$ que aproxime $F(X)$. Esta função $g(\theta, X)$ será posteriormente utilizada para prever quaisquer valores de Y, mesmo que não estejam no conjunto de treino.

A razão pela qual este tipo de Aprendizagem Automática se designa supervisionado é que ao termos os valores de Y podemos supervisionar o processo e estimar o valor de erro

para cada hipótese. A maneira de estimarmos o erro de cada hipótese é através do erro de treino, que classifica quão bem uma hipótese prevê os valores de Y do conjunto de treino.

3.3.2 Algoritmos de Aprendizagem Automática supervisionada

3.3.2.1 Regressão linear

O algoritmo mais simples para resolver problemas de regressão é a regressão linear. Neste algoritmo é escolhida uma linha para prever os valores de Y e a classe da hipótese corresponde ao modelo $y = \theta_1 x_1 + \theta_2 x_2 + ... + \theta_{n+1}$, em que cada x_n corresponde a uma dimensão do espaço de *input*, isto é, o valor de cada *feature*. Na figura 3.1 podemos observar a utilização de polinómios de diferentes graus para prever os valores de y.

Para que a função *g(θ,X)* calculada seja o mais próximo da realidade, isto é, o mais próxima possível de *F(X)*, deve ter o mesmo grau de *F(X)* [4]. Se a função *g(θ,X)* tiver um menor grau de *F(X)* significa que *g(θ,X)* é menos complexa e neste caso temos *underfitting*. À medida que aumentamos o grau de *g(θ,X)* o erro de treino irá diminuir mas se continuarmos a aumentar o grau de *g(θ,X)* chegará uma altura em que a função estará demasiado focada no conjunto de treino. Isto faz com que *g(θ,X)* tenha um erro de treino baixo mas terá um grande erro a prever valores de Y, que não estejam no conjunto de treino, terá um grande *test error*, que é uma estimação do *true error* da função. Neste caso estamos estamos perante *overfitting* e podemos observar isto na figura 3.1, no polinómio de sexto grau.

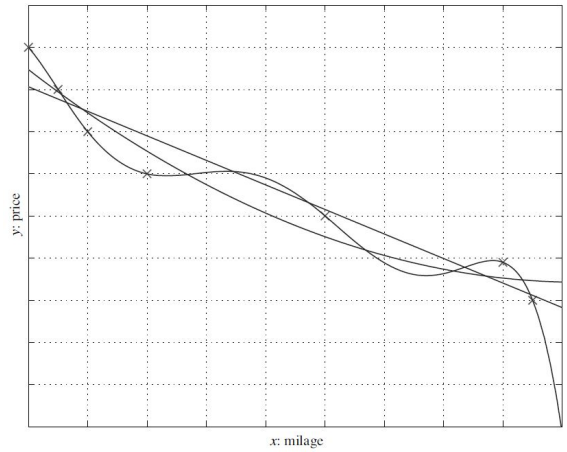


Figura 3.1: Polinómios de primeiro, segundo e de sexto grau utilizados para prever os valores de y. O polinómio de maior grau faz uma previsão perfeita mas pode ocorrer *overfitting* [4]

3.3.2.2 *K-Nearest Neighbor*

Um algoritmo que pode ser usado tanto para classificação como para regressão é o *k-nearest neighbor*. Este algoritmo difere da regressão linear na medida em que neste algoritmo os dados não são utilizados para formular uma hipótese que nos permite perceber

de que maneira os valores de *input* estão relacionados com o valor a prever. Neste algoritmo é utilizado *Lazy Learning*, em vez de *Eager Leaning*, como no caso da Regressão Linear. Nos algoritmos de *Lazy Learning* apenas é efetuada a previsão dos novos valores quando o sistema é questionado acerca desses valores. No algoritmo de *k-nearest neighbor* é consultado o conjunto de treino, especificamente nos k exemplos mais semelhantes ao novo exemplo. Esse exemplo será então classificado conforme a maioria desses k exemplos do conjunto de treino. Uma das decisões a tomar ao utilizar este classificador é que função da distância utilizar. Para *features* com valores numéricos podemos utilizar a *distância de Minkowski*, que é dada pela fórmula:

$$D_{x,x'} = \sqrt[p]{\sum_{d} |x_d - x_d{'}|^p} \tag{3.1}$$

O valor de p também influencia o comportamento deste classificador. Por exemplo, para valores de p entre 0 e 1 as semelhanças numa *feature* tornam-se mais importantes. Na figura 3.2 podemos observar uma comparação de classificadores *k-nearest neighbor* com valores de k diferentes.

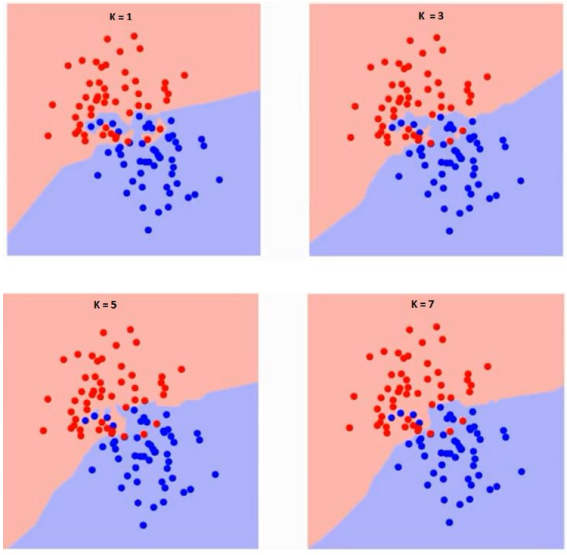


Figura 3.2: Comparação dos classificadores 1-NN, 3-NN, 5-NN e 7-NN para os mesmos dados [41]

O algoritmo de *k-nearest neiqhbor* também pode ser utilizado para regressão através da média dos exemplos no conjunto de treino. O valor previsto será a média dos k exemplos mais semelhantes do conjunto de treino. Contudo, existe uma alternativa melhor chamada *kernel regression* que utiliza uma função contínua que reduz o peso dado aos exemplos do conjunto de treino que são mais diferentes do novo exemplo. Desta forma, é calculada uma média ponderada dos valores no conjunto de treino.

3.3.2.3 Árvores de decisão

Uma árvore de decisão é um modelo hierárquico de Aprendizagem Automática supervisionada em que a região local é identificada numa sequência de divisões recursivas num menor número de passos [4]. Uma árvore de decisão é composta por nós de decisão e nós terminais ou folhas. Num nó de decisão é utilizada uma função de decisão que irá decidir qual dos filhos deste nó será percorrido. Cada nó terminal tem uma etiqueta que em caso de classificação é o código da classe e em caso de regressão é um valor numérico que constitui o *output*. Na tabela 3.3 podemos observar uma árvore de decisão em que os nós ovais são nós de decisão e os nós retangulares são nós terminais.

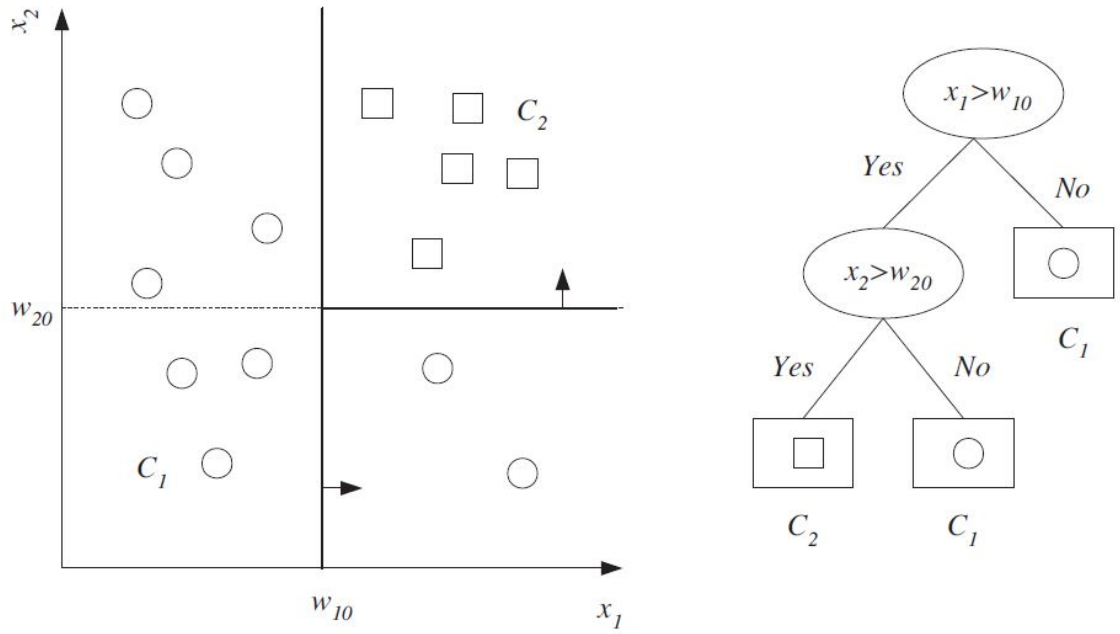


Figura 3.3: Exemplo de um conjunto de dados e a árvore de decisão correspondente [4]

3.3.2.4 *Support Vector Machines*

O objetivo de uma SVM é encontrar o hiperplano que separa duas classes de maneira a podermos efetuar a classificação de novos pontos. Uma SVM procura maximizar a distância entre os pontos mais próximos de classes diferentes, chamada margem. Para efetuar a classificação em casos que não sejam linearmente separáveis é necessário expandir os dados para maiores dimensões e no caso das SVMs são utilizadas *kernel functions*. SVMs também podem ser utilizadas para regressão através de variáveis de folga e de um "tubo"[8], como podemos ver na figura 3.4.

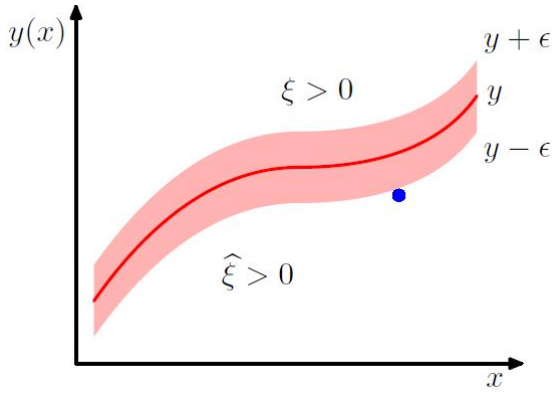


Figura 3.4: Exemplo da utilização de uma SVM para regressão com um "tubo" de diâmetro ϵ. Também podemos ver na figura as várias de folga, ξ e $\hat{\xi}$. Pontos acima do "tubo" têm $\xi > 0$ e $\hat{\xi} = 0$. Pontos abaixo do "tubo" têm $\xi = 0$ e $\hat{\xi} > 0$ [8].

3.3.2.5 *Artificial Neural Network*

Os modelos de ANN são inspirados no cérebro e podem ser utilizados tanto para regressão como classificação [4]. Em alguns domínios o cérebro processa informação de maneira superior aos sistemas atuais e por isso tem um grande interesse económico. Isto faz que haja uma grande vontade em perceber o seu funcionamento para poder ser implementado em computadores e, desta forma, melhorar os sistemas atuais. Ao contrário do computador, que apenas tem um processador, o cérebro tem um número extremamente elevado de unidades de processamento chamados neurónios que operam em paralelo. Os neurónios são responsáveis pelo processamento no cérebro e a memória é armazenada nas sinapses entre os neurónios.

A unidade básica de processamento de uma ANN é o *perceptron*, que podemos observar na figura 3.5. O *output* de um *perceptron* é calculado através da soma dos *inputs* multiplicados pelo seu peso, expresso pela fórmula $y = \sum_{j=1}^{d} w_j x_j + w_0$. w_0 é um valor de interceção para tornar o modelo mais geral, geralmente é utilizado o valor da unidade de *bias* que é sempre +1.

Caso o valor de d seja um o *perceptron* representa uma linha que separa duas classes em caso de classificação. Quanto d tem um valor maior que um o *perceptron* representa um plano. Para decidir a que classe pertence um *input* é preciso implementar uma função de *threshold*. Na figura 3.6 podemos observar uma função *threshold* que decide se *input* pertence à classe C1 ou C2.

Para treinar uma *neural network* é normalmente utilizado *online learning* [4]. Os valores dos pesos são inicialmente gerados aleatoriamente e à medida que são recebidas novas instâncias estes valores são atualizados, sem esquecer as instâncias anteriores.

Um *perceptron* com uma única camada só pode resolver problemas que sejam linearmente separáveis. Para resolvermos os outros problemas é necessário utilizar um Multilayer Perceptron (MLP). Para além disso, um MLP pode ser utilizado para regressão em casos que a função do *input* não é linear. Na figura 3.7 podemos ver um MLP com apenas

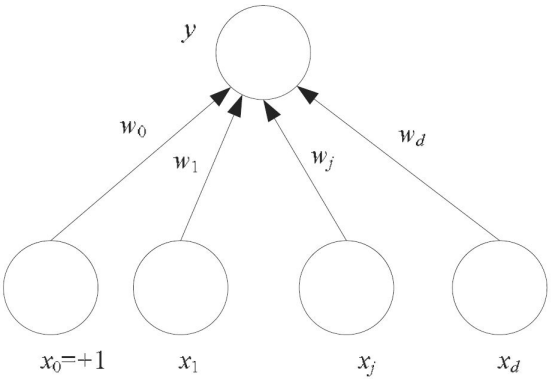


Figura 3.5: *Perceptron* simples. x_j, j = 1, ..., d constituem as unidades de *input*. x_0 é a unidade *bias* que tem sempre o valor 1 e y é *output*. w_j é o peso de cada ligação direta do *input* x_j no *output*. [4]

$$s(a) = \begin{cases} 1 & \text{if } a > 0 \\ 0 & \text{otherwise} \end{cases}$$

then we can

$$\text{choose} \begin{cases} C_1 & \text{if } s(\boldsymbol{w}^T\boldsymbol{x}) > 0 \\ C_2 & \text{otherwise} \end{cases}$$

Figura 3.6: Função *threshold* que decide a classe a partir do *output* [4]

uma camada *hidden*, mas poderia ter várias.

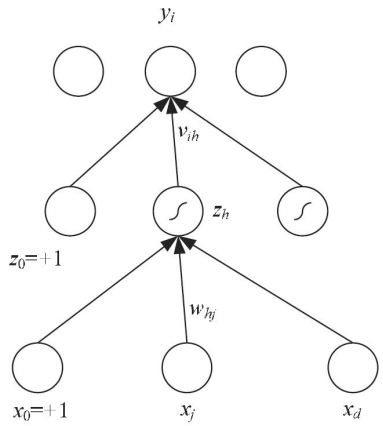


Figura 3.7: Estrutura de um MLP. x_j, j = 0, ..., d constituem as unidades de *input* e z_h, h = 1, ..., H são as unidades *hidden* em que H é a dimensão do *hidden space*. z_0 é o valor do *bias* da camada *hidden*. y_i, i = 1, ..., K são as unidades de *output*. w_{hj} são os pesos na primeira camada e v_{ih} são os pesos da segunda camada [4]

3.3.2.6 Classificador Naive Bayes

Redes Bayesianas são compostas por nós e arcos entre os nós[4]. Cada nó corresponde a uma variável aleatória com um valor da sua probabilidade. Se houver um arco do nó X para o nó Y significa que X tem influência direta em Y. Esta influência é especificada pela

propriedade condicional P(Y|X). Estas dependências podem ser ilustradas por um *Direct Acyclic Graph* (DAG) como podemos observar na figura 3.8 .

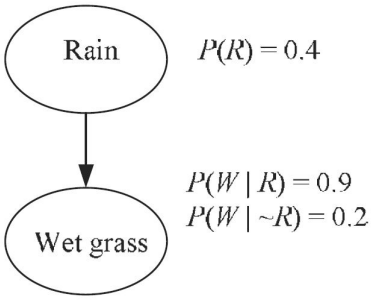


Figura 3.8: Rede bayesiana simples que modela que a chuva é a cause da relva molhada [4]

Um classificador Naive Bayes ignora quaisquer dependências entre as diferentes *features*, o que pode não ser verdade. Contudo este classificador tem bons resultados porque o objetivo consiste em encontrar a classe com maior probabilidade em vez de encontrar a probabilidade absoluta de cada classe.

3.3.3 Aprendizagem Automática não supervisionada

Com Aprendizagem Automática não supervisionada não é possível supervisionar o processo de aprendizagem uma vez que só temos os dados de *input*[4]. O objetivo é encontrar padrões no *input* e perceber o que geralmente acontece[4]. A grande finalidade é transformar os dados de maneira a que sejam mais fáceis de perceber e possam ser utilizados para outros propósitos.

3.3.4 Algoritmos de Aprendizagem Automática não supervisionada

3.3.4.1 Extração de *features*

Esta vertente de Aprendizagem Automática não supervisionada consiste em extrair *features* úteis a partir dos dados. Uma abordagem muito utilizada nesta vertente é a Principal Component Analysis (PCA) que consiste em transformar o conjunto de dados num conjunto ortogonal de coordenadas escolhidas de maneira que os valores ao longo de cada coordenada nova sejam linearmente independentes. PCA tem por base em escolher a direção na qual o conjunto de pontos tem a maior variação, ou seja, a direção na qual estão mais afastados e projetar os dados nessa direção, chamada de *principal component*. Depois são escolhidas novas direções sucessivamente que sejam ortogonais em relação às anteriores e com o mesmo critério da variação.

Outra maneira de projetar um conjunto de dados de grandes dimensões num conjunto de menores dimensões é através de um Mapa Auto-Organizado (SOM). Podemos imaginar um SOM como uma ANN cujos neurónios são dispostos numa matriz de duas dimensões e que cada neurónio tem um vetor de coeficientes com a mesma dimensão que

o conjunto dos dados. Neste algoritmo existem duas medidas de distância, a distância entre dois neurónios e a distância entre um neurónio e um ponto no conjunto dos dados. Inicialmente, são atribuídos valores aleatórios aos coeficientes dos neurónios e em cada iteração é encontrado o neurónio mais próximo de cada ponto, chamado Best Matching Unit (BMU) e o vetor de coeficientes desse neurónio é aproximado desse ponto. Os outros neurónios próximos do BMU também são movidos na mesma direção. Estas alterações são ainda afetadas por uma taxa de aprendizagem que vai diminuindo ao longo do tempo e, por isso, as alterações são cada vez menores.

3.3.4.2 *Clustering*

O objetivo de *clustering* é encontrar *clusters* ou agrupamentos do *input*[4]. Dentro dos *clusters* deve haver um máximo de parecenças e fora dos *clusters* deve haver um mínimo de parecenças. *Clustering* é útil para ajudar a perceber a estrutura dos dados, as relações e também para sumarizar os dados. Várias escolhas devem ser feitas quando queremos agrupar dados e uma delas é escolher o número de *clusters*. Na figura 3.9 podemos observar o resultado de utilizar diferentes números de *clusters*.

Figura 3.9: Utilização de dois, três, quatro e cinco *clusters* para os mesmos dados [23]

3.4 Considerações finais

A utilização de Aprendizagem Automática no *autotuning* de computações em ambientes heterogéneos CPU/GPU é uma área que tem produzido bons resultados nos últimos anos e por é que decidimos utilizar esta abordagem. A técnica de Aprendizagem Automática que vamos utilizar é ANN. No seguinte capítulo vamos falar um pouco mais sobre a nossa solução.

4

Solução

Nesta tese foi desenvolvido um módulo para a *framework* Marrow que decide qual *backend* deve ser utilizado para executar uma dada computação e encontra a melhor configuração para a Unidade de Processamento Gráfico (GPU). Para efetuar esta decisão vamos utilizar um algoritmo de Aprendizagem Automática. Neste capítulo iremos falar sobre as *features* utilizadas (secção 4.2) e a consequente criação do *dataset* (subsecção 4.2.1), a construção das redes neuronais (subsecção 4.2.2) e o procedimento de escolha do *backend* que deve ser utilizado (secção 4.3).

4.1 Visão geral da solução

Na figura 4.1 podemos ver a estrutura atualizada do Marrow e, como é possível verificar, foi acrescentado um módulo à composição inicial do Marrow, presente na figura 2.5. O *expression executor* consulta este módulo de Aprendizagem Automática e o módulo, por sua vez, devolve-lhe as melhores configurações para os executores. O Marrow possui, por exemplo, um executor OpenCL, que executa a computação na GPU, e um executor sequencial, que corre a computação de forma sequencial na Unidade Central de Processamento (CPU). Na figura é ainda possível verificar que o módulo de Aprendizagem Automática possui uma cache. Esta cache contém as configurações pedidas anteriormente e as configurações presentes no *dataset*. Esta cache procura diminuir o tempo de resposta desta nova componente.

A solução desenvolvida tem uma componente inicial que atua em modo *offline*, executada aquando da instalação do sistema ou quando for expressamente pedido, e uma segunda componente executada posteriormente em modo *offline* ou *online*. A primeira componente, executada em *offline*, corresponde à criação do *dataset* e do modelo de Aprendizagem Automática. A razão pela qual o *dataset* tem de ser gerado em todas as máquinas em vez de ser distribuído com o código é porque os valores aí presentes são dependentes de cada máquina. O mesmo se aplica para o modelo de Aprendizagem Automática, uma vez que, este é dependente do *dataset*. A segunda componente, que pode ser executada tanto em *online* como em *offline*, trata-se da obtenção das configurações a partir

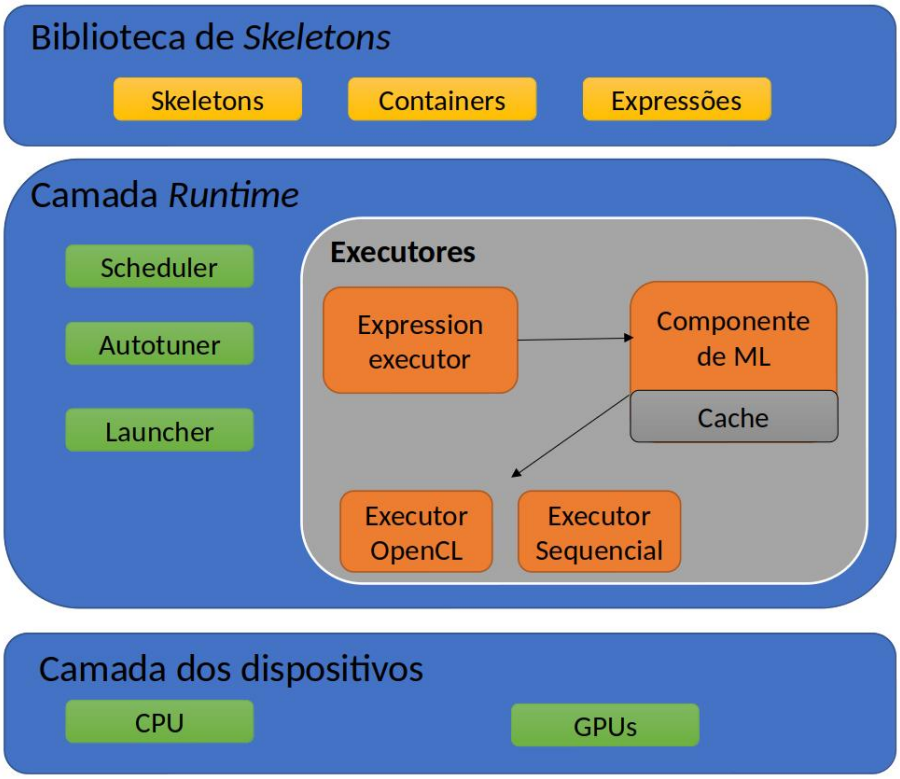


Figura 4.1: Arquitetura do Marrow com o *expression executor*, executores e a componente de Aprendizagem Automática.

das *features* extraídas da computação Marrow. Estas *features* são passadas ao modelo de Aprendizagem Automática e este devolve as configurações. Estas configurações, que serão explicadas com maior detalhe na secção 4.3, correspondem ao *backend* onde deve ser executada a computação, o tempo de execução previsto e o tamanho ideal do bloco de *threads*, caso o *backend* devolvido seja a GPU.

O trabalho desenvolvido pode ser resumido em três programas diferentes, o *dataset generator*, o *autotuner* e a aplicação, representados na figura 4.2. Em *offline* o *dataset generator* cria o *dataset* que será utilizado pelo *autotuner*. No caso do *backend* OpenCL são ainda gerados os ficheiros com os *kernels* OpenCL criados a partir das computações Marrow, caso estes não existam. O *dataset* contém as *features* obtidas de um grande número de computações Marrow e os respetivos valores obtidos através de uma análise da sua execução. Ainda em *offline*, o *autotuner* pega neste *dataset*, cria o modelo de Aprendizagem Automática e guarda-o num ficheiro.

Em *online*, a aplicação utiliza o modelo para efetuar a previsão para novos valores. Caso a aplicação obtenha todas as *features* necessárias para tomar a decisão em tempo de compilação, nomeadamente os tamanhos dos *containers* a serem operados,as decisões são tomadas nesse momento. Caso contrário, as decisões são tomadas em tempo de execução. É passada à aplicação uma computação Marrow para que possa encontrar a melhor configuração. É estimado o tempo de execução para cada *backend* e é devolvido o *backend* com

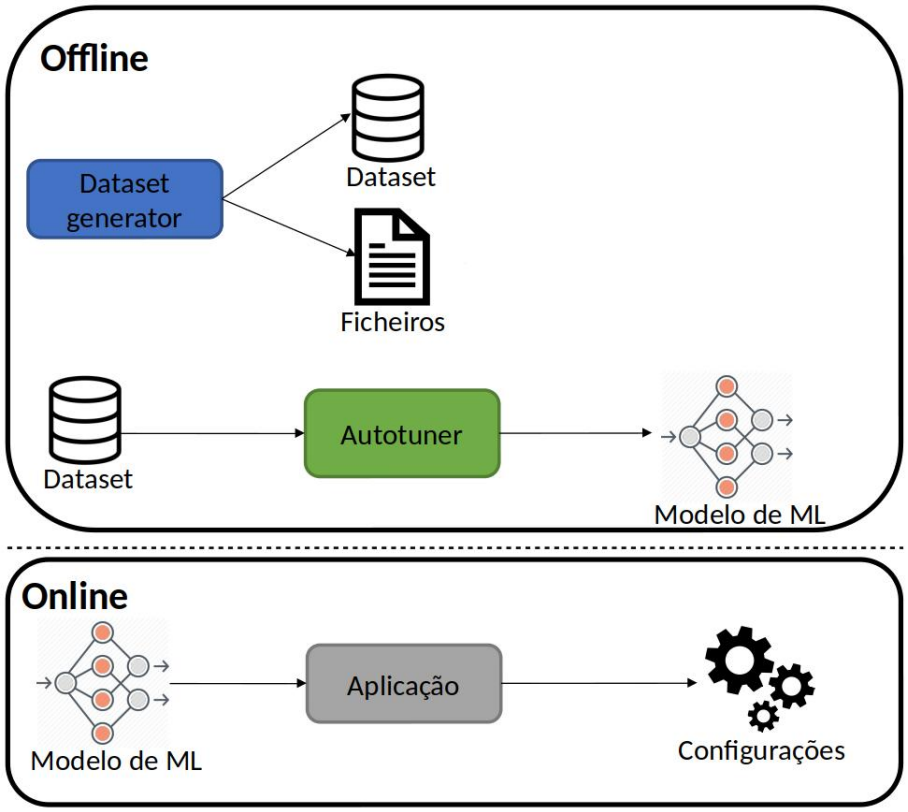


Figura 4.2: Estrutura do trabalho desenvolvido nesta tese. Os diferentes processos encontram-se representados pelas formas retangulares.

o tempo de execução mais baixo. Caso esse *backend* seja o *backend* de OpenCL é devolvido também o tamanho do bloco de *threads* que minimiza o tempo de execução.

4.2 *Features* e construção do *dataset*

Em seguida falaremos sobre as *features* escolhidas e a subsequente construção do *dataset* utilizado no *autotuner*. As *features* são extraídas da árvore gerada pela *framework* Marrow e escolhemos estas *features* porque caracterizam bem cada *kernel*. As *features* são as seguintes:

Reads - número de operações de leitura feitas por cada *kernel* na memória global.

Writes - número de operações de escrita feitas por cada *kernel* na memória global.

Shared reads - número de operações de leitura feitas em memória partilhada por cada *kernel*.

Shared writes - número de operações de escrita feitas em memória partilhada por cada *kernel*.

Operação	Tempo de execução na máquina1 (em nano-segundos)	Tempo de execução na máquina2 (em nano-segundos)
Adição (c = a + b)	8 952 632	1 865 897
Subtração (c = a - b)	8 944 923	1 736 437
Multiplicação (c = a * b)	8 897 594	1 733 457
Divisão (c = a / b)	8 944 458	1 733 551
Módulo (c = a % b)	8 926 687	1 730 421
Comparação (c = a == b)	9 104 696	1 729 356
Potência (c = pow(a, b))	9 140 885	1 785 777
Raiz quadrada (c = sqrt(a))	6 006 538	1 163 729
Seno (c = sin(a))	6 237 901	1 156 710
Expoente de base *e* (c = exp(a))	6 030 313	1 161 155

Tabela 4.1: Tempo de execução de diferentes operações nas duas máquinas testadas. Todas as operações foram efetuadas sobre o mesmo tamanho de *container*.

Allocated memory - tamanho da memória alocada na memória global por cada *kernel* em *bytes*.

Shared memory - tamanho da memória partilhada entre blocos de *threads* em *bytes*.

Há ainda um conjunto de *features* relativas às operações efetuadas em cada computação Marrow. Para termos uma ideia da maneira como as operações influenciam o tempo de execução decidimos efetuar medições para algumas das operações mais comuns. Na tabela 4.1 temos, como exemplo, o tempo de execução dessas operações, para as duas máquinas testadas. Como podemos observar, muitas operações têm tempos de execução semelhantes. Colocámos estes valores numa ferramenta de *clustering* [38] e os resultados estão nas figuras 4.3 e 4.4. As operações foram agrupadas em duas classes consoante o tempo de execução. Agrupar as operações em classes tem a vantagem de simplificar bastante a medição das mesmas na construção do *dataset*, que será discutida em seguida. Isto porque em vez de efetuar as medições para todas as operações, podemos eleger uma operação representante para cada classe e apenas efetuar medições sobre essa operação.

Para este exemplo podemos colocar as operações de adição, subtração, multiplicação, divisão, módulo, comparação e potência na classe 0 e as operações de raiz quadrada, seno e expoente de base *e* na classe 1. Um possível representante para a classe 0 seria a operação de soma e o representante da classe 1 poderia ser a operação de seno. As operações de classe 0 apresentam um tempo de execução cerca de 50% superior às operações de classe 1. O que pode orginar esta diferença é que as operações de classe 0 têm de efetuar o dobro das leituras, uma vez que, têm dois argumentos em vez de um.

Para as duas classes de operações referidas anteriormente teríamos quatro *features* relativas às operações, uma *feature* por cada classe e *features* diferentes para operações sobre inteiros e sobre *floats*. Estas seriam as *features* correspondentes:

Label	Vector	Cluster id	Cluster centroid
ADD	8952632	1	8987410.714285715
SUB	8944923	1	8987410.714285715
MULT	8897594	1	8987410.714285715
DIV	8944458	1	8987410.714285715
MOD	8926687	1	8987410.714285715
COMP	9104696	1	8987410.714285715
POW	9140885	1	8987410.714285715
SQRT	6006538	0	6091584
SIN	6237901	0	6091584
EXP	6030313	0	6091584

Figura 4.3: Resultado do *clustering* [38] das operações para a máquina1.

Label	Vector	Cluster id	Cluster centroid
ADD	1865897	1	1759270.857142857
SUB	1736437	1	1759270.857142857
MULT	1733457	1	1759270.857142857
DIV	1733551	1	1759270.857142857
MOD	1730421	1	1759270.857142857
COMP	1729356	1	1759270.857142857
POW	1785777	1	1759270.857142857
SQRT	1163729	0	1160531.3333333333
SIN	1156710	0	1160531.3333333333
EXP	1161155	0	1160531.3333333333

Figura 4.4: Resultado do *clustering* [38] das operações para a máquina2.

Feature	Árvore 4.5	Árvore 4.6	Árvore 4.7	Árvore 4.8
Reads	2000	2000	3001	3000
Writes	1000	1000	1000	1000
Shared reads	0	0	0	0
Shared writes	0	0	0	0
Allocated memory	8000	8000	12004	12000
Shared memory	0	0	0	0
Class 0 integer ops	1	1	0	0
Class 1 integer ops	0	0	0	0
Class 0 float ops	0	0	3	2
Class 1 float ops	0	1	0	1

Tabela 4.2: Valores das *features* para as diferentes árvores.

Class 0 integer ops - número de operações de classe 0 sobre inteiros realizadas em cada *kernel*.

Class 1 integer ops - número de operações de classe 1 sobre inteiros realizadas em cada *kernel*.

Class 0 float ops - número de operações de classe 0 sobre *floats* realizadas em cada *kernel*.

Class 1 float ops - número de operações de classe 1 sobre *floats* realizadas em cada *kernel*.

Tanto o número de classes de operações, como as operações pertencentes a cada classe, podem ser facilmente alterados que o *dataset generator* e o *autotuner* ajustam-se automaticamente.

Nas figuras 4.5, 4.6, 4.7 e 4.8 temos quatro árvores Marrow como exemplo. Em cada árvore a descrição contém as operações que a originam. Na tabela 4.2 podemos observar os valores das *features* para as respetivas árvores Marrow. As operações de seno, raiz quadrada e expoente de base *e* são contabilizadas como operações sobre *floats* porque

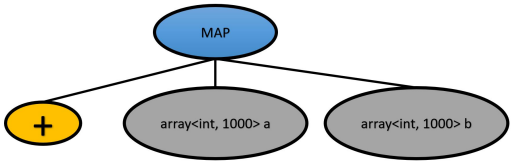


Figura 4.5: c = a + b. Tamanho do *array*: 1000.

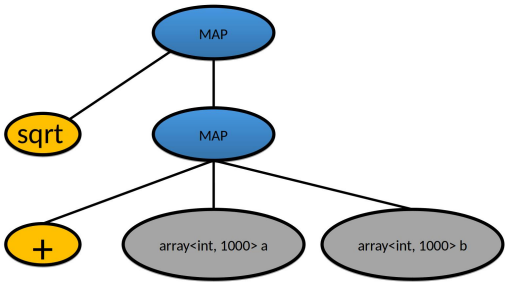


Figura 4.6: c = sqrt(a + b). Tamanho do *array*: 1000.

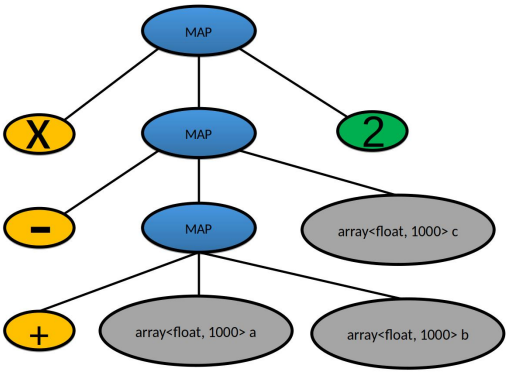


Figura 4.7: d = ((a + b) - c) * 2). Tamanho do *array*: 1000.

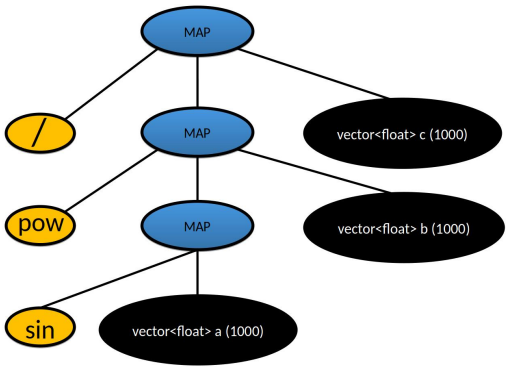


Figura 4.8: d = pow(sin(a), b) / c. Tamanho do *vector*: 1000.

o OpenCL apenas oferece estas operações sobre *containers* do tipo *float*. A árvore 4.5 corresponde a uma soma de dois *arrays* de inteiros cujo tamanho é mil. A árvore 4.6 corresponde à raiz quadrada da soma de *arrays* de inteiros de tamanho mil. A árvore 4.7 corresponde a uma soma de *arrays* de *floats*, uma subtração também de *arrays* de *floats* e a uma multiplicação por dois. O tamanho dos *arrays* é também mil. A árvore 4.8 corresponde ao seno de um *vector* de *floats*, seguida da potência cuja base é o resultado anterior e o expoente é outro *vector* de *floats*. Por último, é feita uma divisão por um *vector* de *floats* adicional. Como os *vectors* são alocados dinamicamente é necessário utilizar *features* parametrizáveis que só podem ser obtidas em tempo de execução.

Nas listagens 3 e 4 podemos ver as principais partes do código relativo à extração da *feature reads* da árvore Marrow. Na listagem 3 podemos observar que é percorrido o lado direito da árvore (linha 17) para calcular o número de leituras. Para determinar o número de escritas é necessário percorrer o lado esquerdo da árvore. Na primeira listagem é ainda chamado o código da segunda listagem com os argumentos da árvore Marrow.

Na listagem 4 são percorridos os diferentes tipos de nós de computação presentes na árvore e consoante o tipo são devolvidos valores diferentes. Estes nós podem ser dos diferentes tipos de *skeletons* do Marrow, como *map*, *scan*, *filter* e *reduce*. Em caso de *array* é simplesmente devolvido o seu tamanho estático (linha 20 da listagem 4) — o tamanho codificado no tipo —, mas no caso de um *container* alocado dinamicamente

Listing 3 Partes essenciais do código no ficheiro reads.hpp. *size_type* é o tipo do tamanho do *container* e *decay* remove os apontadores presentes em *Args*.

```
1  /**
2   * Struct que calcula o número de leituras feitas na memória global
3   *
4   * @tparam AST - árvore Marrow
5   */
6  template<typename AST, typename Enable = void>
7  struct get_reads { };
8
9  /**
10  * @tparam Left - lado esquerdo da árvore Marrow
11  * @tparam Right - lado direito da árvore Marrow
12  */
13 template<typename Left, typename Right>
14 struct get_reads<assignment<Left, Right>> {
15
16     static size_type run(const size_type size) {
17         return get_reads<Right>::run(size);
18     }
19 };
20
21 template<typename AST,  typename... Args>
22 struct get_reads<map_expression<AST, Args...>>{
23
24     static size_type run(const size_type size) {
25         return memory_accesses<decay<Args>...>::run(size);
26     }
27 };
```

isto não pode ser feito. Por isso, para *containers* alocados dinamicamente, como *vectors*, é necessário usar *features* parametrizadas(linha 29 da listagem 4). Neste caso, o parâmetro é o número de elementos do *container* obtido em tempo de execução. Para valores de tipos primitivos, como inteiros, *floats* ou booleanos é devolvido o valor de uma leitura (linha 37 da listagem 4).

4.2.1 Construção do *dataset*

Nesta parte da solução é gerado o *dataset* que será utilizado pelas redes neuronais. No caso do *backend* OpenCL, o executor cria ainda os ficheiros com os *kernels* que serão executados na GPU, caso estes não tenham sido criados anteriormente. Na construção do *dataset* são utilizadas árvores da *framework* Marrow que operam sobre diferentes dimensões de *containers* e com diferentes números de operações. Inicialmente utilizámos dimensões de *containers* que iam de mil inteiros ou *floats* até ao máximo suportado pela GPU, gerando dez valores por cada ordem de grandeza. Por exemplo, na ordem de grandeza de mil eram utilizados os valores de mil, dois mil, três mil, etc., até ao tamanho de nove mil. Na ordem

Listing 4 Partes essenciais do código no ficheiro memory_acesses.hpp. `size_type` representa o tipo do tamanho do `container`. `is_dynamically_managed<geometry_of<C>>::value` verifica se `C` foi alocado dinamicamente, ou seja, se `C` é um `vector`.

```
/**
 * Struct auxiliar que calcula o número de acessos a memória
 *
 * @tparam TypeSequence - sequência de tipos encapsulada no \textit{struct} types
 */
template<typename TypeSequence, typename Enable = void>
struct memory_accesses;

template<typename T, typename... Ts>
struct memory_accesses<types<T, Ts...>> {
    static size_type run(const size_type size) {
        return memory_accesses<T>::run(size) + memory_accesses<types<Ts...>>::run(size);
    }
};

struct memory_accesses<C,
        std::enable_if_t<is_container<C>::value and not is_container_of_container<C>::value and
        not is_dynamically_managed<geometry_of<C>>::value>>  {
    static constexpr size_type run(const size_type) {
        return C::static_size();
    }
};

template<typename C>
struct memory_accesses<C,
        std::enable_if_t<is_container<C>::value and not is_container_of_container<C>::value
        and is_dynamically_managed<geometry_of<C>>::value>>  {
    static size_type run(const size_type size) {
        return size;
    }
};

template<typename T>
struct memory_accesses<T,
        std::enable_if_t<is_marrow_fundamental<T>::value>>  {
    static constexpr size_type run(const size_type ) {
        return 1;
    }
};
```

de dez mil eram utilizados os valores de dez mil, vinte mil, trinta mil, etc., até ao tamanho de noventa mil. E assim sucessivamente, até serem atingidos os tamanhos máximos de *container* suportados pela GPU.

No entanto, esta abordagem faz com que os valores retornados pela rede neuronal (que será explicada na subsecção 4.2.2) para o tempo de execução estejam errados para

Tamanho do *container*	Tempo medido (nano-segundos)	Tempo previsto (nano-segundos)	Erro (%)
1 000	47 293	1 657 920	3 406%
10 000	409 611	2 005 120	390%
100 000	3 989 313	5 480 256	37%
1 000 000	38 832 606	40 340 416	4%
10 000 000	388 655 391	396 463 168	2%
50 000 000	1 969 876 069	1 952 301 184	1%

Tabela 4.3: Erro nas previsões do tempo de execução para a CPU, para diferentes tamanhos de *containers*. É comparado o tempo de execução medido e o valor previsto pela rede neuronal. O erro é calculado da seguinte forma:
$(|tempo\_medido - tempo\_previsto|) \div tempo\_medido$.

Tamanho do *container*	Tempo medido (em nano-segundos)	Tempo previsto (em nano-segundos)	Erro (%)
1 000	100 298	22 160 176	21 994%
10 000	94 939	22 706 640	23 817%
100 000	54 713 529	28 095 984	49%
1 000 000	110 825 425	74 413 696	33%
10 000 000	110 371 960	101 626 992	8%
50 000 000	110 487 861	115 467 352	4%

Tabela 4.4: Erro nas previsões do tempo de execução para a GPU, para diferentes tamanhos de *containers*. É comparado o tempo de execução medido e o valor previsto pela rede neuronal. O erro é calculado da seguinte forma:
$(|tempo\_medido - tempo\_previsto|) \div tempo\_medido$.

os tamanhos de *containers* mais baixos. Tal deve-se à diferença entre os intervalos dos tamanhos de *containers* utilizados, referidos anteriormente. Para os valores mais baixos, o intervalo entre cada medição é de mil, mas para os valores mais altos este intervalo situa-se nas dezenas, centenas ou até nos milhares de milhões. Esta diferença faz com que os valores retornados pela rede para os tamanhos mais baixos sejam mais próximos da média de todos os valores do que dos valores reais. Nas tabelas 4.3 e 4.4 podemos observar os valores medidos do tempo de execução e os valores previstos pela rede neuronal da CPU e pela rede neuronal da GPU. Como é possível comprovar, o erro vai diminuindo à medida que o tamanho do *container* aumenta.

Foram consideradas várias alternativas para resolver este problema. A mais simples seria utilizar tamanhos de *containers* que começassem em mil e fossem incrementados em mil até ser atingido o máximo suportado pela GPU. No entanto, esta opção faria com que o processo de gerar o *dataset* demorasse semanas ou até meses, dependendo da plataforma utilizada.

Outra hipótese que considerámos foi que um erro na normalização dos dados por parte da *framework* utilizada pudesse estar na origem dos erros observados. Por isso, efetuámos a normalização dos dados manualmente e calculámos os erros das previsões.

Rede neuronal	Tempo de treino do *autotuner1* (em micro-segundos)	Tempo de treino do *autotuner2* (em micro-segundos)
CPU	40 678 682	26 719 075
GPU	49 537 009	12 100 062
Total	90 215 691	38 819 137

Tabela 4.5: Comparação dos tempos de treino dos dois *autotuners*. O *autotuner1* apenas utiliza uma rede neuronal para a CPU e outra para a GPU. O *autotuner2* utiliza uma rede neuronal por cada ordem de grandeza, para além de utilizar redes diferentes para CPU e GPU. Para cada valor foram feitas três medições e foi utilizada a média das três medições. No caso do *autotuner2* foi efetuada a soma do tempo de treino de todas as redes neuronais.

No entanto, os erros mantiveram-se e, portanto, concluímos que este não era o problema.

Como será detalhado na subsecção 4.2.2, a solução passou por utilizar uma rede neuronal por cada ordem de grandeza de tamanho do *container*, além de utilizar redes diferentes para CPU e GPU. Tal decisão teve impacto na definição do *dataset*, pois os *datasets* da CPU e GPU foram divididos por cada uma destas ordens para serem utilizados pelas diferentes redes. Por exemplo, para tamanhos de *container* até 9999 é utilizada uma rede neuronal. Para tamanhos entre 10000 e 99999 é utilizada outra rede neuronal, e assim sucessivamente. Na tabela 4.5 podemos observar a comparação do tempo de treino das duas abordagens. Como é possível verificar, o tempo de treino total é mais baixo para o *autotuner* que utiliza várias redes neuronais. Isto permite-nos concluir que o tempo de treino não cresce linearmente com o tamanho do *dataset*, tendo, na realidade, uma complexidade exponencial.

Para avaliar cada um dos *autotuners* comparámos alguns dos valores do tempo de execução presentes nos *datasets* com os valores previstos pelas redes neuronais, para a CPU e para a GPU. Calculámos o erro dividindo a diferença entre o valor real e o valor previsto pelo valor presente no *dataset*. Nas tabelas 4.6 e 4.7 podemos observar a comparação do erro dos dois *autotuners* em relação à previsão do tempo de execução. Como o erro e o tempo de treino são bastante inferiores no *autotuner* que utiliza uma rede neuronal por cada ordem de grandeza, concluímos que esta é a melhor abordagem a utilizar.

Outro aspeto importante é relativo às operações utilizadas por cada tamanho de *container*. Como referimos anteriormente, as operações são agrupadas em classes consoante o seu tempo de execução. O número de classes de operações e o número máximo de operações é passado ao *dataset generator* e este gera todas as combinações disponíveis de número de operações para cada classe. O número máximo de operações é dado como parâmetro, assim como o número de classes de operações. Por exemplo, para o valor predefinido de oito operações o número de operações utilizado varia entre zero e oito, totalizando nove valores diferentes. Para o valor de duas classes de operações dado como exemplo anteriormente, podemos facilmente calcular que existem oitenta e uma combinações diferentes. Apenas são colocadas oitenta combinações no *dataset* porque é excluída a combinação em

Tamanho do *container*	Erro do *autotuner1*(%)	Erro do *autotuner2*(%)
1 000	3 406%	6%
10 000	390%	1%
100 000	37%	4%
1 000 000	4%	6%
10 000 000	2%	4%
50 000 000	1%	1%

Tabela 4.6: Comparação dos erros dos dois *autotuners*. Este erro é para o tempo de execução. *Autotuner1* corresponde ao *autotuner* original e *autotuner2* corresponde ao *autotuner* que utiliza uma rede neuronal por cada ordem de grandeza. Estes erros correspondem aos valores de tempo de execução previstos para a CPU.

Tamanho do *container*	Erro do *autotuner1*(%)	Erro do *autotuner2*(%)
1 000	21 994%	9%
10 000	23 817%	5%
100 000	49%	12%
1 000 000	33%	4%
10 000 000	8%	3%
50 000 000	4%	1%

Tabela 4.7: Comparação dos erros dos dois *autotuners*. Este erro é para o tempo de execução. *Autotuner1* corresponde ao *autotuner* original e *autotuner2* corresponde ao *autotuner* que utiliza uma rede neuronal por cada ordem de grandeza. Estes erros correspondem aos valores de tempo de execução previstos para a GPU.

que ambos os valores de operações são zero. Por cada tamanho de *container* é colocado no *dataset* o dobro deste número de configurações porque são utilizados dois tipos de *container*, *containers* de inteiros e de *floats*. Por isso, com um número máximo de oito operações e duas classes são colocadas cento e sessenta entradas no *dataset* por cada tamanho de *container*. A fórmula que permite determinar o número de combinações que é colocado no *dataset* é a seguinte:

$$2 \times [(Op + 1)^C - 1]$$

em que Op é o número máximo de operações e C é o número de classes de operações.

Além destas configurações é ainda gerado um certo número de configurações aleatórias por cada tamanho de *container*. O número de configurações deste tipo é determinado por um parâmetro passado ao *dataset generator* que corresponde à percentagem de configurações aleatórias adicionais que devem ser geradas. Este parâmetro está predefinido para 100% o que significa que o número de configurações deste tipo é igual ao número de configurações não aleatórias. Em cada uma das configurações aleatórias é produzido um número aleatório para cada uma das classes de operações que representa o número de operações dessa mesma classe. Estes valores são superiores ao número máximo de operações, referido anteriormente, para evitar duplicados no *dataset*.

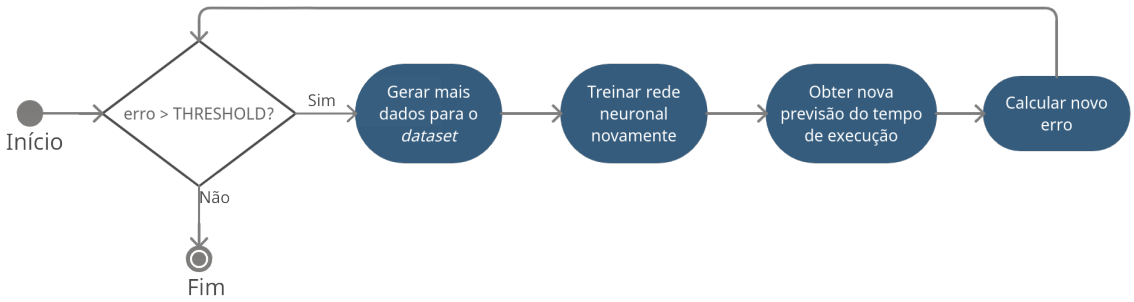


Figura 4.9: Diagrama de atividade simplificado do *feedback loop*. Este ciclo é executado uma vez for cada *backend* a ser considerado.

Para os casos em que as operações só estão definidas para um dado tipo é necessário referir um pequeno pormenor. Por exemplo, no OpenCL, as operações de raiz quadrada, seno e expoente de base *e* só estão definidas para *floats*. Por isso, para efetuar estas operações com *containers* do tipo inteiro é necessário realizar um *cast* de inteiro para *float* em cada *kernel*. Este aspeto é importante porque significa que estas operações são sempre executadas sobre *floats* e são contabilizadas na *feature class 1 float ops*, apesar do tipo do *container* ser inteiro.

Algorithm 1 Pseudocódigo simplificado do *feedback loop* que gera os novos valores para o *dataset*. Em cada iteração é necessário treinar as redes neuronais de novo, para obter os valores atualizados do tempo de execução.

Require: THRESHOLD ▷ Parâmetro que permite determinar quando parar o *feedback loop*
Require: INITIAL_VALUES ▷ Valor ajustável que regula o valor inicial de nr_values_to_generate
Require: MULTIPLIER ▷ Multiplicador ajustável que aumenta nr_values_to_generate
Require: MAX_ITERATIONS ▷ Parâmetro que determina o número máximo de iterações do *feedback loop*

Require: container_size ▷ Tamanho do *container*
Require: features ▷ *Features* extraídas da árvore Marrow
Require: backend ▷ *Backend* para o qual está a ser executado o *feedback loop*
Require: dataset_exectime ▷ Tempo de execução no *dataset* para este conjunto de *features*

$nr\_values\_to\_generate \leftarrow INITIAL\_VALUES$ ▷ Variável que contém o número de novos valores a serem adicionados ao *dataset*
$predicted\_exectime \leftarrow get\_execution\_time(backend, features)$
$error \leftarrow |predicted\_exectime - dataset\_exectime| \div dataset\_exectime$
$iterations \leftarrow 1$ ▷ Variável que contém o número de iterações do *feedback loop*

while $error > THRESHOLD$ & $iterations <= MAX\_ITERATIONS$ **do**
    $generate\_data(backend, container\_size, nr\_values\_to\_generate)$
    $nr\_values\_to\_generate \leftarrow nr\_values\_to\_generate \times MULTIPLIER$
    $predicted\_exectime \leftarrow get\_execution\_time(backend, features)$
    $error \leftarrow |predicted\_exectime - dataset\_exectime| \div dataset\_exectime$
    $iterations \leftarrow iterations + 1$
end while

Um mecanismo que implementámos para reduzir o erro em valores presentes no *dataset* foi um *feedback loop*. Este mecanismo compara o valor previsto pela rede neuronal

para o tempo de execução com o valor presente no *dataset* e, enquanto a diferença for maior que um certo limite, são gerados mais valores para o *dataset*. Os valores gerados são de tamanhos de *containers* semelhantes ao tamanho cujo erro foi detetado, entre este tamanho e o máximo da ordem da grandeza. O processo de criação de novos dados para o *dataset* é explicado com mais detalhe em seguida. Na figura 4.9 temos uma versão simplificada do *feedback loop*, este ciclo é equivalente ao *while* no pseudocódigo 1. É importante notar que a cada iteração do *feedback loop* é gerada uma maior quantidade de dados. No algoritmo 1 a variável que contém o número de tamanhos de *container* diferentes utilizados é $nr\_values\_to\_generate$. O *feedback loop* também tem um parâmetro que regula o número máximo de iterações ($MAX\_ITERATIONS$).

Na linha 6 a função $generate\_data$ gera novos dados para o *dataset* do *backend*, caso ainda não tenha sido atingido o limite de erro escolhido. O processo de criação de novos dados para *backends* GPU está detalhado no algoritmo 2, o único detalhe que está omitido é a escrita no ficheiro. Depois de criados os novos dados, é necessário treinar as redes neuronais novamente para poderem ser obtidas as novas previsões e, de seguida, calculados os novos valores de erro. A função $get\_execution\_time$ corresponde ao processo de treinar a rede neuronal e obter a nova previsão para o *backend* respetivo. No fim de cada iteração é calculado o número de valores a acrescentar ao *dataset*. Tanto o número inicial de valores a gerar como o multiplicador são ajustáveis. Em relação à criação de novos dados para o *dataset*, são gerados *nr_values_to_generate* valores entre *container_size* e o valor máximo da ordem de grandeza de *container_size*, exclusive. Por exemplo, se *nr_values_to_generate* é vinte e *container_size* é dois mil são gerados vinte valores entre mil e dez mil de forma uniforme, excluindo o valor de dez mil. Para cada tamanho de *container* são empregues as mesmas combinações de operações que na criação do *dataset*, incluindo também as configurações aleatórias.

Como as medições de tempos de execução têm uma grande variação, decidimos utilizar o desvio padrão relativo (presente no algoritmo 2 com o nome relative_std_dev) como um indicador da precisão das medições. O desvio padrão relativo é calculado da seguinte maneira:

$$\frac{\text{desvio padrão}}{\text{média}}$$

Na figura 4.10 podemos observar que à medida que são feitas mais medições, o desvio padrão relativo vai baixando e estabilizando. Por isso, para cada *backend* são medidas execuções consecutivas até que o seu conjunto tenha um desvio padrão relativo de menos de 10% (esta constante está presente no algoritmo 2 com o nome MIN_REL_STDDEV), ou seja atingido o número máximo de medições(presente no algoritmo 2 com o nome MAX_MEASURES). Estes valores são ambos parametrizáveis.

O algoritmo 2 é o que encontra o tamanho de bloco ideal para uma certa computação na GPU. Para encontrar as melhores configurações é feita uma pesquisa exaustiva de todos os tamanhos de bloco possíveis para a GPU do sistema, desde o mínimo até ao máximo permitido. Estes valores estão presentes no algoritmo com o nome de *MIN_BLOCK* e

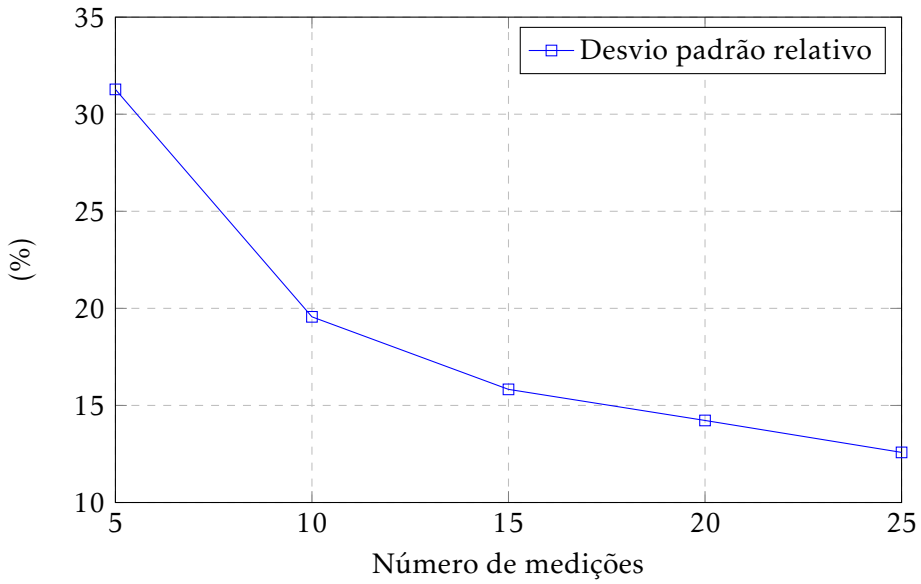


Figura 4.10: Relação entre número de medições e o desvio padrão relativo.

MAX_BLOCK. A variável *relative_std_dev* contém o desvio padrão relativo das medições feitas até ao momento, para um determinado tamanho de bloco. Na linha 10 encontra-se o ciclo *while* que efetua medições até ser atingido o desvio padrão relativo mínimo ou até ser atingido o número máximo de medições. A constante *MIN_MEASURES* representa o valor de medições iniciais e está definido para dois, este é o valor mínimo de medições. É comparada a média das execuções de cada tamanho de bloco e é devolvido o tamanho que minimiza o tempo de execução e o respetivo valor do tempo de execução. Todas as constantes no algoritmo são parametrizáveis.

No entanto, a abordagem de utilizar a mesma rede neuronal para a previsão do tempo de execução na GPU e do tamanho de bloco ideal não produziu bons resultados na componente do tamanho de bloco. Nesta abordagem a rede devolve uma aproximação do tempo de execução e do tamanho de bloco ideal. O problema é que a rede raramente devolve o tamanho de bloco ideal.

Para contrariar tal resultado, decidimos utilizar uma rede apenas para a previsão do tempo de execução da GPU e outra apenas para a previsão do tamanho de bloco ideal. Com a separação das redes, optámos por utilizar uma rede de classificação para o tamanho do bloco, em vez de uma aproximação como para as outras redes. Cada tamanho de bloco corresponde a uma classe. Na próxima subsecção será discutido com maior detalhe a diferença entre as redes.

Na tabela 4.8 temos a comparação dos resultados consequentes da utilização da mesma rede ou redes separadas para a GPU e para o tamanho de bloco. Os valores de diferença presentes na tabela equivalem à diferença entre o tempo de execução resultante do tamanho de bloco ideal e o tempo de execução resultante do tamanho devolvido pela rede. Isto é, em média, o tamanho de bloco devolvido pela rede provoca um tempo de

Algorithm 2 Pseudocódigo simplificado do algoritmo utilizado para encontrar o tamanho de bloco que minimiza o tempo de execução. *execute_kernel(ast, block_size, t)* é uma função que executa a computação Marrow *ast* com o tamanho de bloco *block_size* e guarda os resultados no objeto *t* do tipo Timer. Este objeto é utilizado para obter a média e o desvio padrão das medições efetuadas, assim como o número de medições.

```
Require: MIN_MEASURES      ▷ Número mínimo de medições para cada tamanho de bloco
Require: MAX_MEASURES      ▷ Número máximo de medições para cada tamanho de bloco
Require: MIN_BLOCK         ▷ Tamanho de bloco mínimo
Require: MAX_BLOCK         ▷ Tamanho de bloco máximo
Require: MIN_REL_STDDEV    ▷ Desvio padrão relativo mínimo para terminar as medições
Require: t                 ▷ Objeto do tipo timer utilizado para efetuar as medições

relative_std_dev ← 1   ▷ Variável que contém o desvio padrão relativo do tamanho de bloco que está a ser medido
best_execution_time ← MAX_VALUE   ▷ Variável que contém o melhor tempo de execução, inicializada com o máximo valor permitido para a variável
best_block_size ← 0   ▷ Variável que contém o tamanho de bloco que resulta no melhor tempo de execução

for block_size ∈ [MIN_BLOCK, MAX_BLOCK] do
    t.reset();                                   ▷ Apaga as medições guardadas
    for (i ← 1 to MIN_MEASURES) do
        execute_kernel(ast, block_size, t)
    end for

    relative_std_dev ← t.get_stddev() / t.get_average()

    while (relative_std_dev > MIN_REL_STDDEV &
t.get_number_measurements() < MAX_MEASURES) do
        execute_kernel(ast, block_size, t)
        relative_std_dev ← t.get_stddev() / t.get_average()
    end while

    if (t.average() < best_execution_time) then
        best_execution_time ← t.average()
        best_block_size ← block_size
    end if
end for
```

execução que é 2,5% e 2,4% maior que o tempo de execução mínimo, nos dois casos. Na última linha temos a percentagem de vezes que as duas soluções devolvem o tamanho de bloco ideal. Nos valores das diferenças, as duas abordagens apresentam resultados semelhantes, mas na percentagem de configurações certas a abordagem das redes separadas apresenta resultados significativamente melhores, 17,8% de configurações certas em comparação com os 7,6% da abordagem que utiliza a mesma rede.

4.2.2 Construção da rede neuronal e obtenção do modelo

O algoritmo de Aprendizagem Automática utilizado foi Artificial Neural Networks (ANNs) (referenciado na subsecção 3.3.2.5) por ser uma das técnicas mais utilizadas neste ramo

Resultados	Mesma rede	Redes separadas
Diferença mínima	+0,0%	+0,0%
Diferença máxima	+67,4%	+92,1%
Diferença média	+2,5%	+2,4%
Configurações certas	7,6%	17,8%

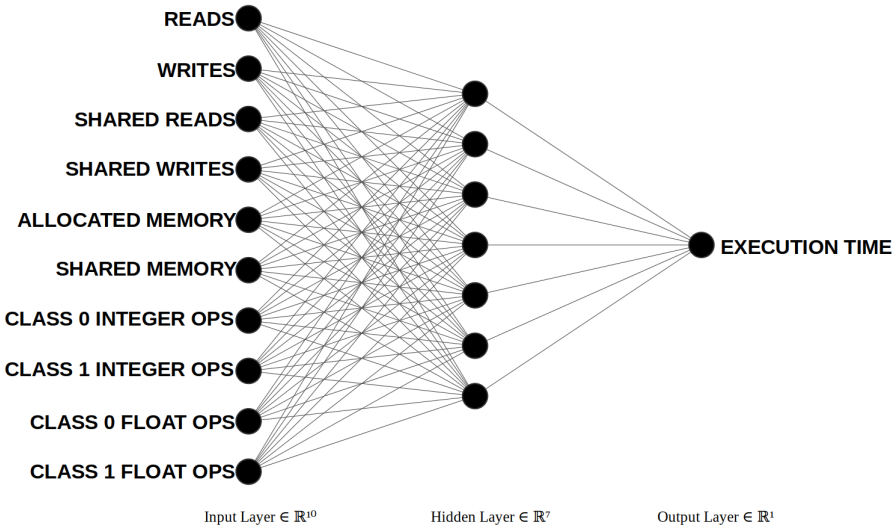
Tabela 4.8: Comparação dos resultados para a previsão do tamanho de bloco decorrentes de utilizar a mesma rede ou redes separadas para a GPU e para o tamanho de bloco. Diferença mínima, máxima e média entre o tempo de execução resultante do tamanho de bloco devolvido pela rede e o tempo de execução derivado do tamanho de bloco ideal. Na última linha temos a percentagem de vezes que a rede devolve o tamanho de bloco ideal. Resultados para valores no *dataset* e para a máquina1.

do *autotuning*, como foi explicado na subsecção 3.1.1.

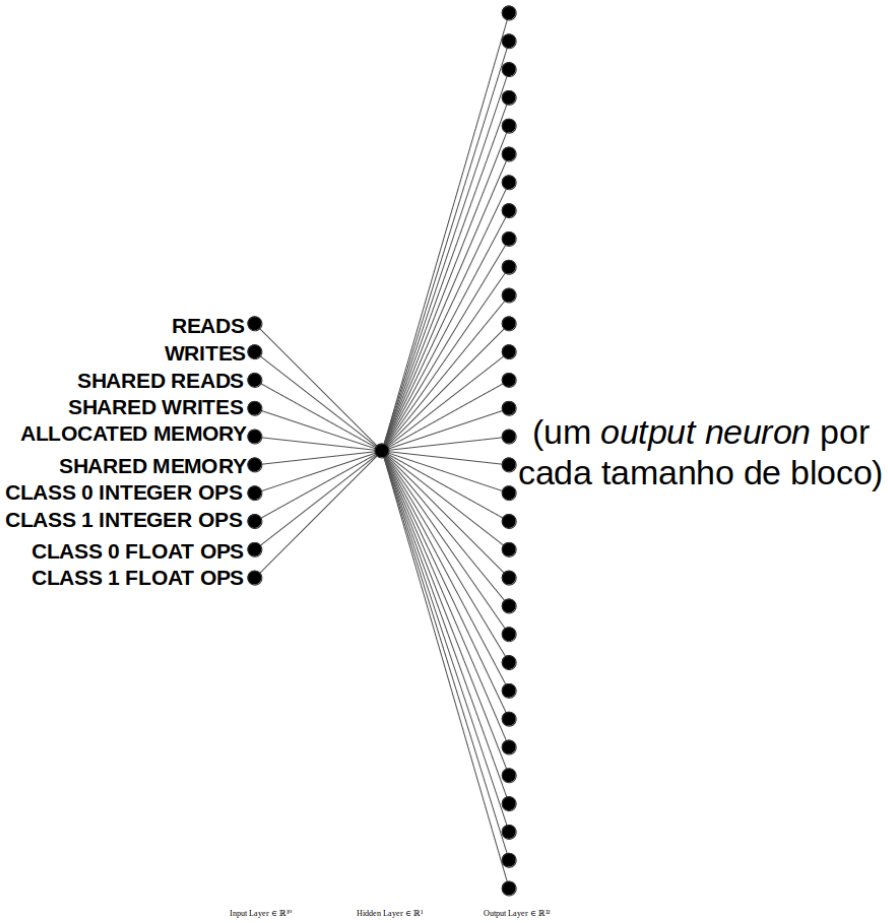
Como aludimos anteriormente, é utilizada uma rede neuronal por cada ordem de grandeza de tamanho de *container*, o que tem a desvantagem de ser necessário treinar várias redes neuronais. Todavia, como o *dataset* é mais pequeno para cada uma das redes, o processo de treino de cada uma é consideravelmente mais rápido. A soma do tempo de treino das redes de todas as ordens de grandeza é também menor que o tempo de treino de uma única rede para todos os tamanhos de *container*, como podemos observar na tabela 4.5. Para além disso, são utilizados dois tipos de redes neuronais, as redes neuronais que prevêem o tempo de execução de execução de um *backend* e as redes neuronais que prevêem o tamanho de bloco de *threads* que minimiza o tempo de execução. Nas redes neuronais que prevêem o tempo de execução é utilizada regressão porque o tempo de execução é um domínio contínuo. Decidimos utilizar uma rede por cada *backend* para evitar a interferência entre dados relativos a execuções nos diferentes *backends*. Por outro lado, a utilização de redes independentes torna o sistema mais modular, uma vez que, permite que sejam considerados outros futuros *backends*, como por exemplo OpenMP ou CUDA. Um pormenor que é necessário referir em relação ao *backend* de OpenCL é que é feita uma previsão para o tempo de execução incluindo a transferência dos dados para a GPU e outra previsão que não inclui esta transferência de dados.

Relativamente às redes neuronais do tamanho de bloco, como estas redes devolvem uma classificação o número de *outputs* é igual ao número de tamanhos de blocos suportados pela GPU do sistema. Cada *output* é um valor entre zero e um que corresponde à probabilidade do *input* pertencer a essa classe. O tamanho de bloco devolvido pela rede é aquele cujo *output* apresenta o maior valor. Utilizámos classificação para este tipo de redes porque o número de tamanhos de bloco suportados pela GPU é um valor discreto.

Na figura 4.11 podemos ver o esquema de uma das redes que prevê o tempo de execução de uma computação Marrow num determinado *backend* e uma das redes do tamanho de bloco. Cada rede do tempo de execução utiliza um *dataset* diferente com os valores do respetivo *backend*. A dimensão da camada de *input* é dez para todas as redes porque possuem dez *inputs*. Em relação à camada *hidden*, optámos por sete *hidden neurons* para as



(a) Previsão do tempo de execução



(b) Previsão do tamanho de bloco

Figura 4.11: Em cima temos o tipo de rede utilizada para prever o tempo de execução de um *backend*. Em baixo temos uma das redes neuronais para o tamanho de bloco, estas redes prevêem o tamanho de bloco ideal [35].

redes do tempo de execução e um *hidden neuron* para as redes do tamanho do bloco. Escolhemos estes valores porque a biblioteca que utilizámos possui uma funcionalidade que permite encontrar o número ideal de *hidden neurons* e estes foram os valores devolvidos. Outra diferença entre as redes neuronais está na camada de *output*. Nas redes neuronais do tempo de execução esta camada tem um de dimensão. Nas redes do tamanho de bloco esta camada tem uma dimensão igual ao número de tamanhos de bloco diferentes suportado pela GPU. Como podemos observar na figura 4.11a as redes do tempo de execução apenas têm um *output* que corresponde ao tempo de execução do respetivo *backend* e a rede do tamanho de bloco tem trinta e dois *outputs* que é o número de tamanhos de blocos suportados pelos dois sistemas em que efetuámos os testes.

A biblioteca utilizada para a definição e execução da rede neuronal foi a biblioteca OpenNN [30]. Optámos por utilizar esta biblioteca porque, segundo a informação recolhida *online*, esta é a biblioteca para C++ que tem a melhor performance o que é muito importante nesta tese.

A fase de treino é precedida pela normalização dos valores das *features*. Este processo é importante porque torna o treino da rede neuronal mais rápido e faz com que os *outputs* sejam mais corretos.

O método de otimização utilizado no treino da rede foi o Adaptive Moment Estimation (ADAM) [22]. Definimos como número máximo de épocas (*epochs*) do algoritmo o valor de cem mil, em que uma *epoch* corresponde a uma iteração do *dataset*. Optámos por este valor de *epochs* porque é suficiente grande para efetuar o processo de treino, independentemente das configurações escolhidas na criação do *dataset*. Depois do treino, a rede neuronal é guardada num ficheiro XML para ser utilizada na próxima vez que seja necessária.

4.3 Escolha do *Backend* em Tempo de Execução

Para decidir em que *backend* deve ser executada uma computação e quais as melhores configurações é necessário passar a árvore Marrow à aplicação. A partir dessa árvore são extraídas, em tempo de compilação, as *features* para que a respetiva rede neuronal possa efetuar a previsão. Contudo, no caso de *containers* alocados dinamicamente, as *features* parametrizáveis não são obtidas em tempo de compilação, uma vez que estas dependem de parâmetros que só são obtidos em tempo de execução.

Para cada rede neuronal, a aplicação verifica se já existe um ficheiro com essa mesma rede, se existir importa a rede do ficheiro. Se o ficheiro não existir e a plataforma Marrow estiver configurada para fazer uso do *autotuning*, o *autotuner* efetua o treino da rede neuronal e guarda-a no ficheiro.

Uma vez as redes treinadas e os modelos computados, a aplicação obtém as previsões necessárias e verifica qual das previsões qual dos *backends* tem a previsão mais baixa.

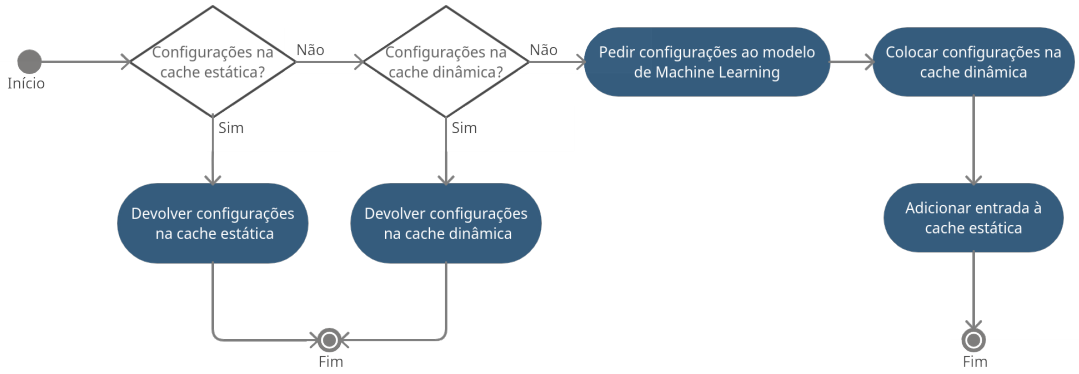


Figura 4.12: Diagrama de atividade do processo de consulta e atualização das caches.

Para otimizar o tempo de resposta da aplicação só é feita a previsão do tamanho de bloco se a GPU tiver a previsão mais baixa do tempo de execução. Tomámos esta decisão porque o valor devolvido para o tamanho de bloco só será utilizado nestas condições e, ainda porque, o treino da rede neuronal do tamanho de bloco é o mais demorado, devido ao superior número de *outputs* desta rede.

Em seguida, a aplicação devolve a plataforma que apresenta o menor tempo de execução e as respetivas configurações, se necessário. O valor é devolvido sob a forma de um `std::array` de inteiros com três valores. O primeiro valor é o menor tempo de execução, em nano-segundos, entre as previsões do tempo de execução dos *backends*. O segundo representa o *backend* ideal previsto e é utilizado um enumerado para identificar cada *backend*. O último valor das configurações diz respeito ao tamanho ideal do bloco de *threads*, quando o dispositivo ideal previsto é a CPU este valor é zero. Este valor de *output* é ainda colocado na cache, que será discutida em seguida.

4.3.1 Caches

A aplicação possui duas caches, uma estática e uma dinâmica. Estas caches possuem as configurações devolvidas anteriormente pela aplicação. A cache estática funciona à base de ficheiros C++ incluído no código da aplicação e a cache dinâmica funciona à base de um mapa na aplicação. Implementámos estas caches para evitar algumas consultas das redes neuronais e, desta forma, diminuir o tempo de resposta da aplicação. Na figura 4.12 temos o diagrama de atividade que representa o processo de consulta e atualização das caches. Em primeiro lugar, verificamos se as configurações se encontram na cache estática ou na cache dinâmica. Caso estejam, são devolvidas as configurações. Caso as configurações não se encontrem nas caches é necessário utilizar os modelos de Aprendizagem Automática para obter as configurações e, em seguida, estas configurações são colocadas nas caches.

4.3.1.1 Cache Estática

Em relação à cache estática, através da utilização dos *templates* do C++ conseguimos embutir a cache no código da aplicação. Utilizámos *structs* com *template parameters* para representar as entradas na cache, escritos num ficheiro C++ com um nome parametrizável. Cada *struct* contém o tempo de execução da computação, o dispositivo onde deve ser executada a computação e o tamanho ideal do bloco de *threads*. Foram utilizados dois *template parameters*, em que o primeiro é relativo ao *backend* e o segundo corresponde ao tipo da árvore Marrow. Este tipo é composto pelas operações presentes na árvore e os *containers* sobre os quais são feitas essas operações. Por exemplo, para a árvore Marrow $c = a + b$, em que os *containers* a, b e c têm o tipo `array<int, 1000>`, o tipo da árvore é o seguinte:

```
assignment<
   array<int, 1000>,
   map_expression<
      call<plus<int, int>>,
      array<int, 1000>&,
      array<int, 1000>&>>
```

em que, como descrito na secção 4.2, as operações de *map* correspondem a operações entre *containers* e as operações de *assignment* correspondem à atribuição de valores a *containers*.

Agora vamos mostrar de que maneira são representadas as entradas na cache estática Esta é a entrada que é devolvida por omissão:

```
template<backend Backend, typename AST, typename CodeOptimizationStages>
struct static_cache_entry<execution<Backend, AST, CodeOptimizationStages>> {
    static constexpr bool cached = false;
    static constexpr long long exectime = 0;
    static constexpr marrow::backend backend = Backend;
    static constexpr int block_size = 0;
};
```

Este é o *struct* que será retornado se uma certa árvore não estiver presente na cache. O primeiro campo com o nome *cached* indica se uma dada computação está presente na cache. Como esta é a entrada devolvida por omissão o valor é *false*, em todas as outras entradas o valor é *true*. O campo *exectime* corresponde ao tempo de execução da computação e o seu valor por omissão é zero. Em seguida, temos o campo *backend* que corresponde ao dispositivo onde deve ser executada a computação, *marrow::backend::CPP* representa a CPU e *marrow::backend::OpenCL* representa a GPU. O valor por omissão é o mesmo que o *template parameter* passado à cache com o nome *Backend*. Por último, o campo *block_size* representa o tamanho do bloco de *threads*. O valor por omissão é zero. Quando o dispositivo é a CPU o campo *block_size* é omitido.

Listing 5 Cache estática com duas entradas de exemplo, para além da entrada *default*. Ambos os *structs* correspondem à operação $c = a + b$. No primeiro estes *containers* têm mil de tamanho e no segundo têm dez mil de tamanho. *marrow* :: *backend* :: *CPP* corresponde ao *backend* da CPU e *marrow* :: *backend* :: *OpenCL* corresponde ao backend da GPU.

```
1  template<backend Backend, typename AST, typename CodeOptimizationStages>
2  struct static_cache_entry<execution<Backend, AST, CodeOptimizationStages>> {
3      static constexpr bool cached = false;
4      static constexpr long long exectime = 0;
5      static constexpr marrow::backend backend = Backend;
6      static constexpr int block_size = 0;
7  };
8
9  template<>
10 struct static_cache_entry<execution<marrow::backend::CPP, assignment<array<int, 1000>,
   ↪ map_expression<call<plus<int, int>>, array<int, 1000>&, array<int, 1000>&>>>>{
11         static constexpr bool cached = true;
12         static constexpr long long exectime = 6038;
13         static constexpr marrow::backend backend = marrow::backend::CPP;
14 };
15
16 template<>
17 struct static_cache_entry<execution<marrow::backend::OpenCL, assignment<array<int,
   ↪ 10000>, map_expression<call<plus<int, int>>, array<int, 10000>&, array<int,
   ↪ 10000>&>>>>{
18         static constexpr bool cached = true;
19         static constexpr long long exectime = 39255;
20         static constexpr marrow::backend backend = marrow::backend::OpenCL;
21         static constexpr int block_size = 640;
22 };
```

Na listagem 5 podemos observar duas entradas de exemplo na cache estática, para além da entrada por omissão que já foi referida. Os dois *structs* a seguir ao *struct* de omissão correspondem a árvores Marrow que executam a operação $c = a + b$ mas sobre tamanhos de *containers* diferentes. O primeiro corresponde a uma árvore que opera sobre um *container* de tamanho mil. Os respetivos valores presentes na cache são que o tempo de execução é de 6038 nano-segundos e o dispositivo onde deve ser executada a computação é a CPU. O segundo *struct* corresponde a uma árvore que opera sobre um *container* de tamanho dez mil. Os valores presentes na cache são que o tempo de execução é de 39255 nano-segundos, o dispositivo onde deve ser executada a computação é a GPU e o tamanho de bloco ideal é 640.

4.3.1.2 Cache Dinâmica

A cache dinâmica consiste num mapa que contém as configurações ideais para uma determinada computação Marrow. Como chave do mapa utilizámos o *hash code* do tipo da computação, porque é eficiente como chave em mapas e causa muito poucas colisões.

O *hash code* é obtido da seguinte maneira:

```
const size_t hash_code = typeid(AST).hash_code();
```

AST é o tipo da computação Marrow. Os valores nos mapas são as configurações sob a forma de um *std* :: *array* de inteiros com três valores descritos anteriormente. Este mapa não tem tamanho predefinido e permite continuar a adicionar entradas até que seja atingido o limite do sistema ou da biblioteca C++.

4.3.1.3 Colocação das configurações nas caches

Em relação à transferência dos valores de configurações para as caches, o processo é explicado em seguida. Quando é instanciado um objeto da aplicação é inicializada a cache dinâmica e é deixada vazia. À medida que são feitos pedidos à aplicação esta verifica se as configurações pretendidas estão presentes em alguma das caches. Caso estejam, são devolvidas as configurações em cache. Caso contrário, é pedido à rede neuronal que faça as previsões e essas configurações são guardadas na cache dinâmica, antes de serem retornadas pela aplicação. Quando for chamada a função *destructor* da aplicação é percorrida a cache dinâmica e são colocadas todas as entradas aí presentes em *structs* no ficheiro cache.hpp que contém a cache estática.

```
#if __has_include(<marrow/autotuning/cache.hpp>)
#include <marrow/autotuning/cache.hpp>
#endif
```

O código acima verifica em tempo de compilação se o ficheiro *cache.hpp* existe e, em caso afirmativo, inclui-o no código. No entanto, quando o ficheiro é criado é necessário forçar o *rebuild* da aplicação, para que o ficheiro *cache.hpp* possa ser incluído na próxima execução da aplicação. Por isso, implementámos um mecanismo que força o *rebuild* alterando muito ligeiramente o ficheiro C++ que contém o código da aplicação. Nas alterações seguintes ao ficheiro não é necessário forçar o *rebuild* porque este já é efetuado automaticamente, uma vez que, o ficheiro já está incluído no código.

Uma otimização que decidimos efetuar foi colocar todos os valores do *dataset* na cache estática. Isto significa que quando são pedidas as configurações para valores de *features* que estão no *dataset*, as configurações devolvidas pela aplicação são reais e não previsões.

4.3.2 Comentários finais

Neste capítulo apresentámos a solução desenvolvida que utiliza Aprendizagem Automática para decidir em que *backend* deve ser executada uma computação e encontra o tamanho de bloco ideal. Enunciámos as *features* usadas e explicámos o processo de criação do *dataset*, o qual sofreu alterações ao longo do tempo para colmatar os problemas encontrados. Explicámos também o processo de criação das redes neuronais que, da mesma

forma, foi sofrendo alterações pelas mesmas razões. Por fim, esclarecemos o método de escolha do *backend* a utilizar e descrevemos as duas caches.

No próximo capítulo vamos proceder à avaliação da solução desenvolvida, apresentando os resultados para os dois sistemas e sob diferentes parâmetros. Vamos também avaliar o tempo de resposta da aplicação e o *speedup* resultante das caches.

5

Avaliação Experimental

Neste capítulo vamos proceder à avaliação do trabalho realizado. Nesta avaliação vamos utilizar o *backend* sequencial, que executa as computações na Unidade Central de Processamento (CPU), e o *backend* OpenCL, que executa as computações na Unidade de Processamento Gráfico (GPU). Na secção 5.1 começamos por apresentar as questões a que queremos responder com a nossa avaliação. De seguida, descrevemos a metodologia e o ambiente de testes nas secções 5.2 e 5.3, respetivamente. Na secção 5.4 temos as respostas às perguntas referidas anteriormente e na secção 5.5 temos as considerações finais.

5.1 Objetivos

Nesta secção vamos apresentar as diferentes perguntas a que é necessário responder relativamente à avaliação do trabalho realizado. Estas perguntas podem ser agrupadas em dois tipos distintos. O primeiro tipo de perguntas está relacionado com a precisão das previsões feitas pela solução desenvolvida. As perguntas são as seguintes:

Qual é a precisão da previsão do tempo de execução na CPU? Uma das perguntas a que vamos responder é relativa à precisão das previsões do tempo de execução para a CPU. Vamos avaliar as previsões para valores presentes dentro e fora do *dataset* de treino apresentando a percentagem de erro. Doravante o *dataset* de treino será simplesmente referido como *dataset*. Vamos também mostrar alguns exemplos de previsões feitas pela rede neuronal e comparar com os valores reais, mostrando a percentagem de erro.

Qual é a precisão da previsão do tempo de execução na GPU? Outra pergunta a que vamos responder corresponde à precisão das previsões do tempo de execução na GPU. Vamos avaliar estas previsões da mesma maneira que as previsões para a CPU.

Qual é a precisão para a escolha do *backend* correcto? Vamos também verificar se a aplicação escolheu o *backend* certo para efetuar a computação. Para tal, vamos consultar os valores reais no *dataset* do tempo de execução e verificar se a aplicação devolveu o dispositivo correto. Por exemplo, para a computação c = a + b com um *container* de

tamanho mil o tempo de execução para a máquina1 na GPU é de 43 999 nano-segundos e na CPU é de 40 651 nano-segundos. Logo, a plataforma devolvida pela aplicação deve ser a CPU porque tem um tempo de execução mais baixo.

Qual é a precisão da classificação para o tamanho do bloco de *threads*? Outro dos aspetos em que vamos avaliar o trabalho desenvolvido é em relação à classificação do tamanho de bloco. As previsões também serão avaliadas para valores dentro e fora do *dataset* mas não será calculada a percentagem de erro. Em vez disso, será calculada a percentagem de vezes que o tamanho de bloco previsto é o tamanho de bloco ideal. Para além disso, será calculada a diferença mínima, máxima e média entre o tempo de execução resultante do tamanho de bloco ideal e o tempo de execução resultante do tamanho de bloco desenvolvido. Por exemplo, se o tamanho de bloco ideal produz um tempo de execução de 50 000 nano-segundos e o tamanho de bloco previsto produz um tempo de execução de 60 000 nano-segundos a diferença será 20% porque 60 000 é 20% maior que 50 000.

O segundo tipo de perguntas prende-se com o tempo de resposta do trabalho desenvolvido. As perguntas são as seguintes:

Qual o tempo de criação do *dataset*? A componente mais demorada da solução desenvolvida é a criação do *dataset*. Vamos mostrar de que maneira a duração deste processo varia consoante o tamanho do *container* e apresentar a fórmula que permite estimar o tempo total da criação do *dataset*.

Qual o tempo de resposta da aplicação? Por último, vamos avaliar a aplicação em termos do seu tempo de resposta, que depende de uma série de fatores. Mesmo que o processo de treino das redes neuronais já tenha sido concluído existem outros fatores que o podem influenciar. Se as configurações para determinados valores de *features* já foram calculadas previamente só é necessário consultar uma das caches, que apresentam tempos de resposta diferentes entre si e que serão comparados. Caso contrário, é necessário pedir à rede neuronal que faça uma previsão para estes valores e, em seguida, colocá-los na cache. Este processo de consulta da rede neuronal também depende se a rede já se encontra em memória ou é necessário importar a partir do ficheiro XML.

5.2 Metodologia de avaliação

No cálculo da percentagem de erro vamos ter em conta as decisões tomadas pela aplicação, a previsão do tempo de execução e qual o tamanho de bloco indicado, no caso da plataforma indicada ser a GPU. Vamos executar a aplicação cinco vezes para cada exemplo e, de seguida, calcular a média dessas cinco previsões. Esta média será utilizada nos exemplos de previsões na secção 5.4.

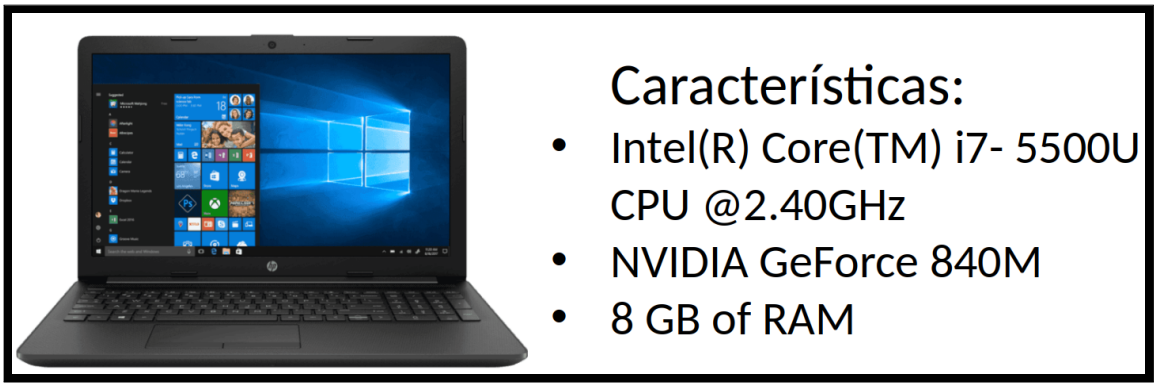


Figura 5.1: Computador pessoal utilizado [20] com as respetivas características. O sistema operativo desta máquina é Ubuntu 20.04.3 LTS, a versão do compilador g++ é 9.3.0 e a versão do OpenCL é 2.2. Será referido como **máquina1**.

Para proceder à avaliação das previsões para valores no *dataset* são iterados todos os valores aí presentes e são comparados com os valores previstos pela rede neuronal. Em relação à avaliação para valores fora do *dataset*, são gerados números aleatórios utilizados como tamanho de *container* e outros números aleatórios para o valor de operações de cada classe. Estes valores de operações são superiores aos presentes no *dataset*. São comparados os valores devolvidos pelas redes com os valores reais, que têm de ser medidos pois não se encontram no *dataset*, e é calculada a percentagem de erro. A percentagem de erro será calculada com a seguinte fórmula:

$$\frac{|valor\ \ previsto\ -\ valor\ \ real|}{valor\ \ real} \times 100$$

5.3 Ambiente de testes

Esta tese foi testada num computador pessoal e no *cluster* disponibilizado pelo Departamento de Informática da FCT/UNL. Nas figuras 5.1 e 5.2 podemos observar as duas máquinas onde foram feitos os testes e as respetivas características. Como podemos verificar o *cluster* é composto por vinte e três nós com características distintas. Os nós do *cluster* que utilizámos para testes foram os nós dezasseis a dezoito.

5.4 Resultados

Depois de realizados os testes supra-referidos passamos a analisar as respostas obtidas.

5.4.1 Qual é a precisão da previsão do tempo de execução na CPU?

Nas tabelas 5.1 e 5.2 temos a comparação de algumas previsões das redes neuronais com os valores reais no *dataset*. Nestas tabelas podemos imediatamente verificar que os resultados para a máquina1 foram melhores que os resultados para a máquina2. Uma possível explicação para o erro mais elevado na máquina2 é a aparente instabilidade das medições

Características

Machine	CPU	Total Cores/Threads	Memory	Network	Main Disk (/, /tmp)	Extra Disks (/mnt/localhddX)	Graphics
node1-5	AMD EPYC 7281	16/32	128 GiB DDR4 2666 MHz	2 x 10 Gbps	1.8 TB HDD	-	-
node6-8	2 x Intel Xeon E5-2620 v2	12/24	64 GiB DDR3 1600 MHz	2 x 1 Gbps	100 GB SSD	930 GB HDD	-
node9-11	Intel Xeon X3450	4/8	8 GiB DDR3 1333 MHz	2 x 1 Gbps	230 GB HDD	230 GB HDD	-
node12	4 x AMD Opteron 6272	32/64	64 GiB DDR3 1600 MHz	2 x 1 Gbps	120 GB SSD	2x 460 GB HDD	-
node13	8 x AMD Opteron 8220	16/16	27 GiB	2 x 1 Gbps	130 GB HDD	-	-
node14	Intel Xeon E5-2603 v2	4/4	16 GiB DDR3 1333 MHz	2 x 1 Gbps	930 GB HDD	-	NVIDIA GeForce GTX 1050 Ti
node15	Intel Xeon E5-2620 v2	6/12	32 GiB DDR3 1600 MHz	2 x 1 Gbps	110 GB HDD	-	-
node16-18	2 x Intel Xeon E5-2609 v4	16/16	32 GiB DDR4 2400 MHz	2 x 1 Gbps	110 GB SSD	-	NVIDIA Quadro M2000
node19-23	2 x AMD Opteron 2376	8/8	16 GiB DDR2 667 MHz	2 x 1 Gbps	150 GB HDD	-	-

Figura 5.2: Imagem de um *Cluster* [31] e características do *cluster* utilizado para testes composto por vinte e três nós e as características de cada nó [43]. O sistema operativo destes nós é Debian GNU/Linux 11, a versão do compilador g++ é 10.2.1 e a versão do OpenCL é 3.0. Será referido como **máquina2**.

no *dataset*. Para os valores na tabela 5.1 podemos observar que medições para o mesmo tamanho de *container* apresentam valores muito semelhantes. Por exemplo, nas medições com tamanho de *container* de cem mil, na tabela 5.1 os valores são 4 002 041 e 4 009 320 nano-segundos. Contudo, para a tabela 5.2 estes valores são 518 501 e 426 753 nano-segundos, ou seja, apresentam uma diferença muito maior entre si. Esta instabilidade dos valores presentes no *dataset* pode explicar o erro mais elevado para a máquina2.

Nas tabelas 5.3 5.4 podemos observar os resultados para valores dentro e fora do *dataset*. Como é possível constatar, a diferença de erro entre as duas máquinas manteve-se, com a máquina1 a apresentar um erro médio de 1,4% e 2,9% e a máquina2 a apresentar um erro médio de 11,5% e 15,5%. Nas figuras 5.3 e 5.4 temos ainda a comparação entre as previsões e os valores reais do tempo de execução na CPU, para as duas máquinas.

5.4.2 Qual é a precisão da previsão do tempo de execução na GPU?

Nas tabelas 5.5 e 5.6 temos os resultados para algumas previsões do tempo de execução na GPU, para as duas máquinas. Nas tabelas 5.7 5.8 podemos observar a percentagem de erro para valores dentro e fora do *dataset*, respetivamente. Como podemos verificar, os resultados obtidos foram bastante bons. A percentagem de erro média para valores no *dataset* foi de 3,1% e 3,3% para as duas máquinas. Mas o aspeto mais interessante são as previsões para valores fora do *dataset*, que apresentaram uma percentagem de erro de 1,1% e 2,2% para as duas máquinas. A razão pela qual os valores exteriores ao *dataset* apresentam melhores resultados pode estar relacionado com o o tamanho de *container*

Operações	Tamanho do *container*	Tempo de execução no *dataset* (ns)	Tempo de execução previsto (ns)	erro (%)
b = sin(a)	1 000	40 651	40 808	0,4%
d = sin((a + b) - c)	1 000	40 421	40 881	1,1%
g = pow((a * b) / c, exp(sqrt(d / e) + f))	1 000	40 690	41 172	1,2%
c = a + b	10 000	397 974	402 218	1,0%
e = ((a + b) / c) * d	10 000	397 553	400 907	0,8%
b = exp(sin(sqrt(a)))	100 000	4 002 041	4 147 968	3,6%
d = a / (sin(a) + sqrt(b % c))	100 000	4 009 320	4 086 589	1,9%
d = (a + b) - c	1 000 000	38 676 072	39 516 528	2,1%
c = exp(a)	1 000 000	39 280 221	41 384 976	5,3%
j = pow(((((a + b) - c) * d) / e), sqrt(sin(f % exp(g))))	10 000 000	388 565 734	392 232 576	0,9%

Tabela 5.1: Comparação entre os valores do tempo de execução para a CPU no *dataset* e os valores previstos pela rede neuronal. Resultados para a máquina1.

Operações	Tamanho do *container*	Tempo de execução no *dataset* (ns)	Tempo de execução previsto (ns)	erro (%)
b = sin(a)	1 000	2 877	4 271	48,4%
d = sin((a + b) - c)	1 000	3 710	4 407	18,8%
g = pow((a * b) / c, exp(sqrt(d / e) + f))	1 000	3 700	4 659	25,9%
c = a + b	10 000	40 507	60 494	49,3%
e = ((a + b) / c) * d	10 000	51 279	54 841	6,9%
b = exp(sin(sqrt(a)))	100 000	283 443	365 127	28,8%
d = a / (sin(a) + sqrt(b % c))	100 000	380 037	385 331	1,4%
d = (a + b) - c	1 000 000	3 566 267	3 661 544	2,7%
c = exp(a)	1 000 000	2 902 287	3 160 196	8,8%
j = pow(((((a + b) - c) * d) / e), sqrt(sin(f % exp(g))))	10 000 000	35 790 315	36 221 104	1,2%

Tabela 5.2: Comparação entre os valores do tempo de execução para a CPU no *dataset* e os valores previstos pela rede neuronal. Resultados para a máquina2.

Erro	Máquina1	Máquina2
Erro mínimo	0,0008%	0,0006%
Erro máximo	42,5%	78,9%
Erro médio	1,4%	5,9%
Desvio padrão	1,6%	8,8%

Tabela 5.3: Erro médio, mínimo, máximo e desvio padrão do erro das previsões do tempo de execução para valores no *dataset* e para a CPU. Resultados para a máquina1 e máquina2.

Erro	Máquina1	Máquina2
Erro mínimo	0,008%	2,4%
Erro máximo	42,7%	28,8%
Erro médio	2,9%	19,9%
Desvio padrão	3,7%	2,8%

Tabela 5.4: Erro médio, mínimo, máximo e desvio padrão do erro das previsões do tempo de execução para valores fora do *dataset* e para a CPU. Resultados para a máquina1 e máquina2.

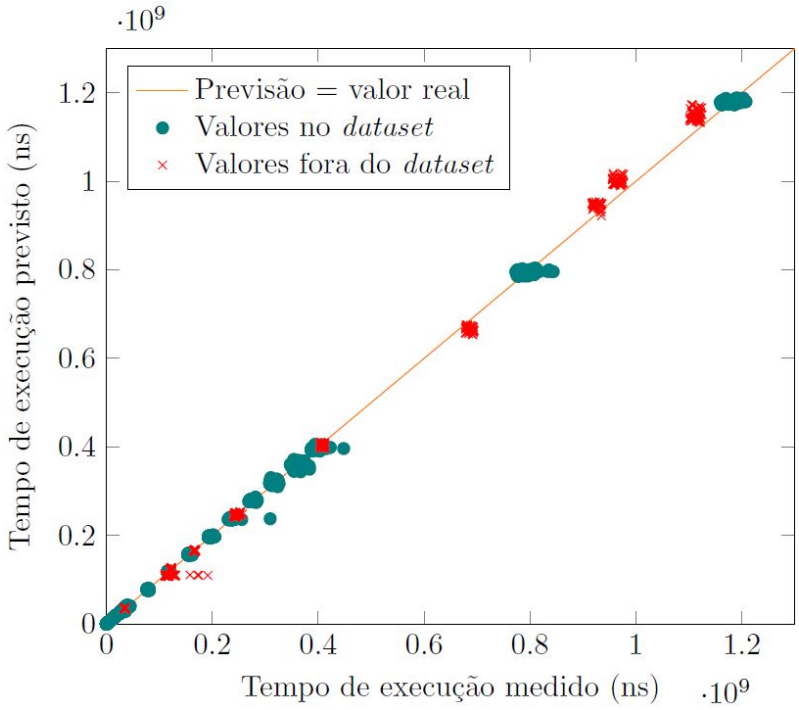


Figura 5.3: Comparação entre as previsões e os valores reais para o tempo de execução na CPU. Resultados para a máquina1.

utilizado nesta avaliação. Como os valores são gerados aleatoriamente, o tamanho de *container* médio utilizado nesta avaliação será superior ao tamanho empregue na avaliação do *dataset*. Este pormenor pode afetar os resultados porque quanto maior o tamanho de *container* melhores aparentam ser os resultados, ao olharmos para as tabelas de exemplos 5.5 e 5.6. Nas figuras 5.5 e 5.6 temos ainda a comparação entre as previsões e os valores reais do tempo de execução na GPU, para as duas máquinas.

5.4.3 Qual é a precisão para a escolha do *backend* correcto?

Nesta secção vamos avaliar a performance da aplicação, no que diz respeito à previsão do *backend* que minimiza o tempo de execução. Nas tabelas 5.9 e 5.10 temos diferentes exemplos de previsões feitas pela rede neuronal e as respetivas avaliações, para a máquina1 e máquina2. Na tabela 5.11 temos o resultado das previsões para todos os valores presentes

Operações	Tamanho do *container*	Tempo de execução no *dataset* (ns)	Tempo de execução previsto (ns)	erro (%)
b = sin(a)	1 000	43 999	43 792	0,5%
d = sin((a + b) - c)	1 000	76 208	50 701	33,4%
g = pow((a * b) / c, exp(sqrt(d / e) + f))	1 000	63 411	62 546	1,4%
c = a + b	10 000	48 290	52 983	9,7%
e = ((a + b) / c) * d	10 000	53 704	53 748	0,1%
b = exp(sin(sqrt(a)))	100 000	80 729	89 309	10,6%
d = a / (sin(a) + sqrt(b % c))	100 000	188 667	187 057	0,8%
d = (a + b) - c	1 000 000	1 340 314	1 328 550	0,9%
c = exp(a)	1 000 000	690 112	656 192	4,9%
j = pow(((((a + b) - c) * d) / e), sqrt(sin(f % exp(g))))	10 000 000	22 726 946	22 681 156	0,2%

Tabela 5.5: Comparação entre os valores do tempo de execução para a GPU no *dataset* e os valores previstos pela rede neuronal. Resultados para a máquina1.

Operações	Tamanho do *container*	Tempo de execução no *dataset* (ns)	Tempo de execução previsto (ns)	erro (%)
b = sin(a)	1 000	42 861	41 112	4,1%
d = sin((a + b) - c)	1 000	44 669	43 317	3,0%
g = pow((a * b) / c, exp(sqrt(d / e) + f))	1 000	51 059	48 590	4,8%
c = a + b	10 000	39 255	37 521	4,4%
e = ((a + b) / c) * d	10 000	37 413	39 959	6,8%
b = exp(sin(sqrt(a)))	100 000	49 156	49 261	0,2%
d = a / (sin(a) + sqrt(b % c))	100 000	65 444	66 149	1,1%
d = (a + b) - c	1 000 000	281 873	278 437	1,2%
c = exp(a)	1 000 000	159 896	187 526	17,3%
j = pow(((((a + b) - c) * d) / e), sqrt(sin(f % exp(g))))	10 000 000	5 029 862	5 050 018	0,4%

Tabela 5.6: Comparação entre os valores do tempo de execução para a GPU no *dataset* e os valores previstos pela rede neuronal. Resultados para a máquina2.

Erro	Máquina1	Máquina2
Erro mínimo	0,0%	0,0005%
Erro máximo	51,4%	42,9%
Erro médio	3,1%	3,3%
Desvio padrão	5,5%	4,1%

Tabela 5.7: Erro médio, mínimo, máximo e desvio padrão do erro das previsões do tempo de execução para valores no *dataset* e para a GPU. Resultados para a máquina1 e máquina2.

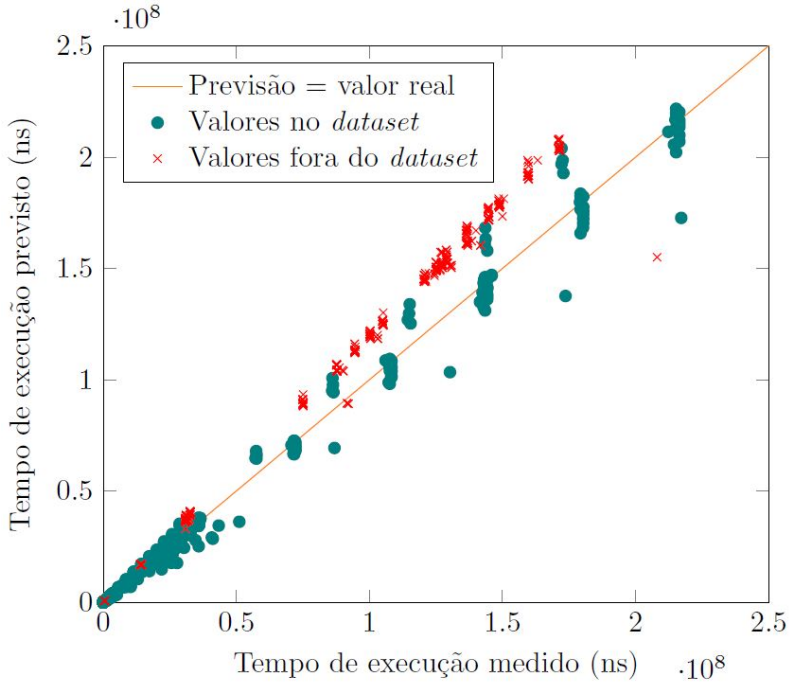


Figura 5.4: Comparação entre as previsões e os valores reais para o tempo de execução na CPU. Resultados para a máquina2.

Erro	Máquina1	Máquina2
Erro mínimo	0,009%	0,002%
Erro máximo	3,3%	21,8%
Erro médio	1,1%	2,2%
Desvio padrão	0,9%	3,3%

Tabela 5.8: Erro médio, mínimo, máximo e desvio padrão do erro das previsões do tempo de execução para valores fora do *dataset* e para a GPU. Resultados para a máquina1 e máquina2.

no *dataset* e para valores fora do *dataset*. Como podemos observar nesta e nas tabelas anteriores, as previsões apresentam resultados bastante bons. Apesar dos resultados para o tempo de execução apresentarem um erro que consideramos normal, o seu impacto é muito baixo porque as redes têm uma eficácia acima de 97,9% na escolha do *backend*. Uma particularidade dos resultados é que os as previsões para valores no *dataset* são piores que as previsões para valores fora do mesmo. Tal como nas previsões do tempo de execução na GPU, isto pode estar relacionado com o tamanho de *container* gerado aleatoriamente uma vez que, a maior parte das previsões erradas ocorre para tamanhos de *container* mais baixos e a probabilidade desses valores serem gerados aleatoriamente é bastante baixa.

5.4.4 Qual é a precisão da classificação para o tamanho do bloco de *threads*?

Tal como referimos anteriormente, para avaliar os resultados das previsões para o tamanho de bloco seguimos uma abordagem diferente das outras avaliações. Para além de

Operações	Tamanho do *container*	Dispositivo previsto	Avaliação
b = sin(a)	1000	CPU	✓
d = sin((a + b) - c)	1000	CPU	✓
g = pow((a * b) / c, exp(sqrt(d / e) + f))	1000	CPU	✓
c = a + b	10000	GPU	✓
e = ((a + b) / c) * d	10000	GPU	✓
b = exp(sin(sqrt(a)))	100000	GPU	✓
d = a / (sin(a) + sqrt(b % c))	100000	GPU	✓
d = (a + b) - c	1000000	GPU	✓
c = exp(a)	1000000	GPU	✓
j = pow(((((a + b) - c) * d) / e), sqrt(sin(f % exp(g))))	10000000	GPU	✓

Tabela 5.9: Avaliação das previsões da aplicação para o dispositivo onde deve ser executada a computação. Resultados para a máquina1.

Operações	Tamanho do *container*	Dispositivo previsto	Avaliação
b = sin(a)	1000	CPU	✓
d = sin((a + b) - c)	1000	CPU	✓
g = pow((a * b) / c, exp(sqrt(d / e) + f))	1000	CPU	✓
c = a + b	10000	GPU	✓
e = ((a + b) / c) * d	10000	GPU	✓
b = exp(sin(sqrt(a)))	100000	GPU	✓
d = a / (sin(a) + sqrt(b % c))	100000	GPU	✓
d = (a + b) - c	1000000	GPU	✓
c = exp(a)	1000000	GPU	✓
j = pow(((((a + b) - c) * d) / e), sqrt(sin(f % exp(g))))	10000000	GPU	✓

Tabela 5.10: Avaliação das previsões da aplicação para o dispositivo onde deve ser executada a computação. Resultados para a máquina2.

Máquina	Distribuição correta para valores no *dataset* (%)	Distribuição correta para valores fora do *dataset* (%)
Máquina1	99,9%	100 %
Máquina2	97,9%	100 %

Tabela 5.11: Resultados para a distribuição das computações pela CPU e GPU, tanto para valores no *dataset* como para valores fora do *dataset*.

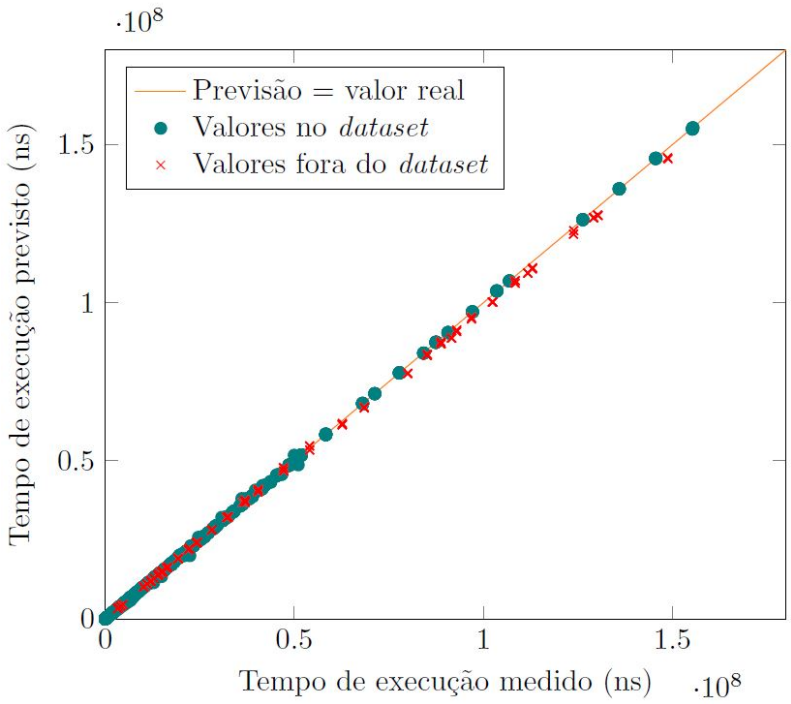


Figura 5.5: Comparação entre as previsões e os valores reais para o tempo de execução na GPU. Resultados para a máquina1.

calcularmos a percentagem de vezes que a rede neuronal devolve o tamanho de bloco ideal, vamos apresentar os valores da diferença mínima, máxima e média entre o tempo de execução resultante do tamanho de bloco ideal e o tempo de execução resultante do tamanho de bloco devolvido pela rede.

Na tabela 5.12 temos os resultados para a máquina1. A diferença média é de 2,4% e 0,6% para valores dentro e fora do *dataset*, respetivamente. Isto significa que o tamanho de bloco devolvido pela rede resulta num tempo de execução que, em média, é 2,4% e 0,6% maior que o ideal. Ainda nesta tabela podemos observar que a rede neuronal devolveu o tamanho de bloco ideal em 17,8% e 7,1% das medições.

Na tabela 5.13 estão os resultados para a máquina2. Esta máquina apresenta melhores resultados que a anterior com uma diferença média de 1,8% e 0,3%. A percentagem de configurações certas é de 25,1% e 52,3%, o que são resultantes muito bons.

5.4.5 Qual o tempo de criação do *dataset*?

Na figura 5.7 temos um gráfico que mostra a relação entre o tamanho de *container* e o tempo de criação dos dados para esse mesmo tamanho. Como podemos verificar, o tempo de criação dos dados cresce de forma linear em relação ao tamanho de *container* utilizado. Desta maneira, podemos estimar o estimar o tempo total de criação do *dataset* com a seguinte fórmula:

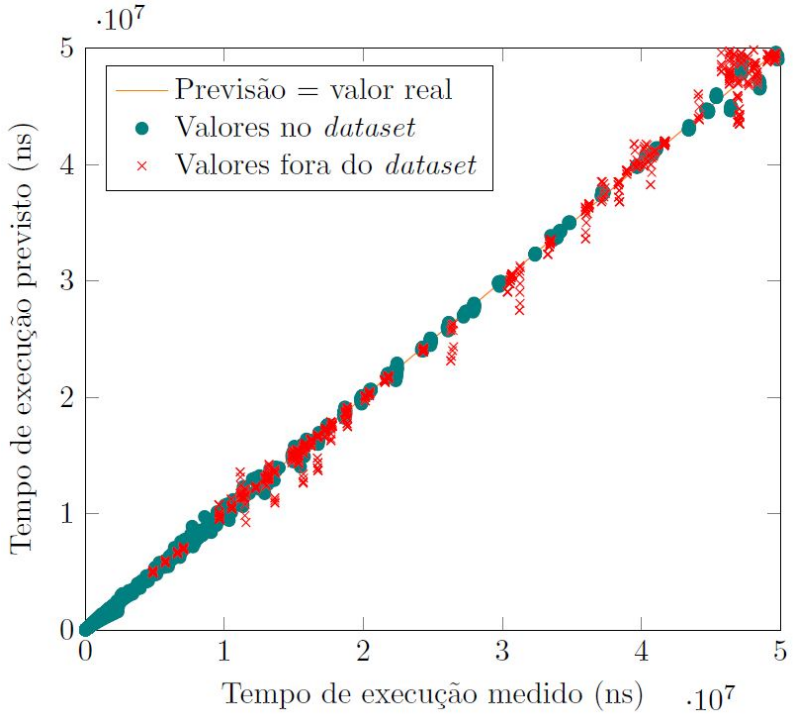


Figura 5.6: Comparação entre as previsões e os valores reais para o tempo de execução na GPU. Resultados para a máquina2.

Resultados	Valores no *dataset*	Valores fora do *dataset*
Diferença mínima	+0,0%	+0,0%
Diferença máxima	+92,1%	+12,4%
Diferença média	+2,4%	+0,6%
Configurações certas	17,8%	7,1%

Tabela 5.12: Resultados das previsões do tamanho de bloco para valores dentro e fora do *dataset*. Diferença mínima, máxima e média entre o tempo de execução resultante do tamanho de bloco devolvido pela rede e o tempo de execução derivado do tamanho de bloco ideal. Na última linha temos a percentagem de vezes que a rede devolve o tamanho de bloco ideal. Resultados para a máquina1.

$$T_{total} = \sum_{i=m}^{M} [\, (i \,/\, m) \times t_m \,]$$

Em que m é o tamanho mínimo de *container* utilizado na construção do *dataset*, M é o tamanho de *container* máximo suportado pelo sistema e t_m é o tempo de criação dos dados para o *container* m. É importante referir que os tamanhos de *container* utilizados na construção do *dataset* são valores discretos e são explicados na secção 4.2.1.

Resultados	Valores no *dataset*	Valores fora do *dataset*
Diferença mínima	+0,0%	+0,0%
Diferença máxima	+29,1%	+13,2%
Diferença média	+1,8%	+0,3%
Configurações certas	25,1%	52,3%

Tabela 5.13: Resultados das previsões do tamanho de bloco para valores dentro e fora do *dataset*. Diferença mínima, máxima e média entre o tempo de execução resultante do tamanho de bloco devolvido pela rede e o tempo de execução derivado do tamanho de bloco ideal. Na última linha temos a percentagem de vezes que a rede devolve o tamanho de bloco ideal. Resultados para a máquina2.

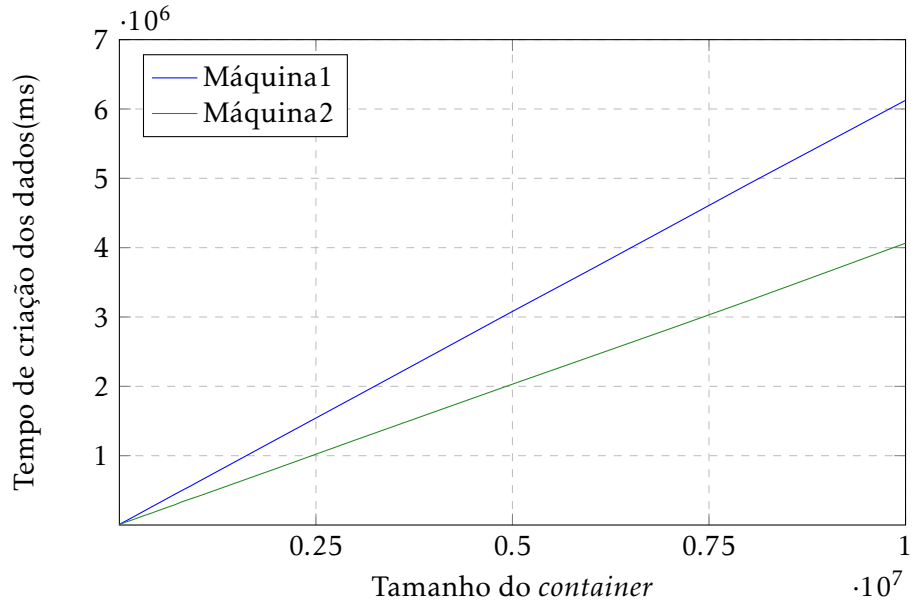


Figura 5.7: Relação entre o tamanho do *container* e a duração do *dataset* generator. Resultados para a máquina1 e para a máquina2.

5.4.6 Qual o tempo de resposta da aplicação?

Em primeiro lugar vamos apresentar os tempos de treino das redes neuronais feitos no *autotuner* e, em seguida, vamos apresentar os tempos de resposta da aplicação. A razão pela qual o tempo de treino das redes neuronais está separado do tempo de resposta da aplicação é porque o treino das redes é feito em *offline* e a aplicação é executada em *online*, como é possível constatar na figura 4.2.

Nas tabelas 5.14 e 5.15 temos o tempo de treino das redes neuronais com ou sem *feedback loop* nas duas máquinas. A primeira linha das tabelas é referente ao caso em que é necessário efetuar o treino de uma rede da CPU, uma rede da GPU e uma rede do tamanho de bloco. Para além disso, é efetuada uma iteração do *feeback loop*. Na segunda linha são treinadas as três redes neuronais referidas anteriormente mas não é feita qualquer iteração do *feedback loop*. Na última linha apenas é treinada uma rede da CPU e uma rede da GPU. Como podemos verificar, existe uma grande diferença no tempo de treino de duas ou três

Fatores	Tempo de treino (micro-segundos)
Treino das três redes neuronais e *feedback loop*	2 797 785 133
Treino das três redes neuronais	2 079 805 394
Treino de duas redes neuronais	186 941 896

Tabela 5.14: Tempos de treino das redes neuronais. Para todos os valores foram feitas várias medições e foi utilizada a média das várias medições. Resultados para a máquina1.

Fatores	Tempo de treino (micro-segundos)
Treino das três redes neuronais e *feedback loop*	62 856 749
Treino das três redes neuronais	48 584 021
Treino de duas redes neuronais	5 844 397

Tabela 5.15: Tempos de treino das redes neuronais. Para todos os valores foram feitas várias medições e foi utilizada a média das várias medições. Resultados para a máquina2.

redes neuronais. Esta diferença existe porque o treino da rede do tamanho de bloco é o mais demorado, devido ao maior número de *outputs* desta rede.

Nas tabelas 5.16 e 5.17 temos os tempos de resposta da aplicação caso as redes neuronais já tenham sido treinadas. Na primeira linha é necessário carregar os modelos das redes neuronais a partir dos ficheiros e calcular os *outputs*. Este tempo inclui o processo de importar as três redes neuronais e calcular os três *outputs*. Na segunda linha as redes neuronais já estão em memória e é apenas necessário calcular o *outputs*. Este tempo também inclui o processo de importar as três redes neuronais e calcular os três *outputs*. Nas duas últimas linhas, apenas é necessário consultar as caches para obter as configurações. Como é possível observar, ambas as caches diminuem drasticamente o tempo de execução. A cache estática diminui o tempo de resposta para aproximadamente zero porque as configurações ficam disponíveis em tempo de compilação.

5.5 Considerações finais

Com os resultados aqui apresentados, podemos concluir que a solução desenvolvida funciona bastante bem. Os dois principais objetivos desta tese prendiam-se com a escolha do *backend* para executar uma computação e, caso esse *backend* fosse a GPU, encontrar o tamanho do bloco de *threads* que minimizasse o tempo de execução. Estes objetivos foram claramente alcançados. A escolha do *backend* apresentou uma eficácia acima de 97% em todos os testes efetuados, em ambas as máquinas testadas. Relativamente ao tamanho do

Fatores	Tempo de resposta (micro-segundos)
Importar redes neuronais dos ficheiros e calcular *outputs*	15771
Calcular *output*	162
Consulta da cache dinâmica	3
Consulta da cache estática	≈ 0

Tabela 5.16: Tempos de execução da aplicação nas diferentes circunstâncias. Para todos os valores foram feitas várias medições e foi utilizada a média das várias medições. Resultados para a máquina1.

Fatores	Tempo de resposta (micro-segundos)
Importar redes neuronais dos ficheiros e calcular *outputs*	39255
Calcular *outputs*	228
Consulta da cache dinâmica	1
Consulta da cache estática	≈ 0

Tabela 5.17: Tempos de execução da aplicação nas diferentes circunstâncias. Para todos os valores foram feitas várias medições e foi utilizada a média das várias medições. Resultados para a máquina2.

bloco, a aplicação devolve, em média, um tamanho de bloco que resulta num tempo de execução que fica a menos de 2,4% do tempo de execução mínimo. Este valor atinge os 0,6% e 0,3% nos valores fora do *dataset* nas duas máquinas testadas. Para além disso, a percentagem de vezes que é devolvido o tamanho de bloco ideal é bastante satisfatória.

Os resultados obtidos na previsão dos tempos de execução na CPU e na GPU foram também muito positivos, com exceção das previsões da CPU para uma das máquinas, que apresentou resultados razoáveis.

No próximo capítulo vamos expor as conclusões desta tese e falar sobre o trabalho futuro que pode vir a ser realizado.

6

Conclusões e trabalho futuro

Neste capítulo vamos apresentar as conclusões do trabalho desenvolvido. Vamos analisar se foram cumpridos os objetivos a que nos propusemos, expor algumas das dificuldades encontradas e propor trabalho que pode ser realizado no futuro.

6.1 Conclusões

Esta tese teve como objetivo acrescentar um módulo à *framework* Marrow que decidisse em que *backend* deve ser executada uma computação e, no caso de esse *backend* ser a Unidade de Processamento Gráfico (GPU), obter ainda o tamanho do bloco de *threads* que minimizasse o tempo de execução. Para alcançarmos este objetivo utilizámos um algoritmo de Aprendizagem Automática chamado Artificial Neural Networks (ANNs). A imprevisibilidade deste tipo de algoritmos fez com que o número de redes neuronais utilizadas e os *datasets* utilizados pelas mesmas fossem sofrendo alterações ao longo do tempo.

No entanto, os resultados resultados obtidos forem muito positivos. Relativamente à escolha do *backend*, a solução apresenta uma eficácia superior a 97%, para ambas as máquinas. Em relação ao tamanho do bloco, o valor devolvido para esta configuração resulta num tempo de execução que fica, em média, a menos de 2,4% do tempo resultante do tamanho de bloco ideal. Acresce ainda que a percentagem de vezes que o tamanho de bloco ideal é devolvido é bastante satisfatória. Apesar de este não ser o principal foco desta tese, os valores previstos para o tempo de execução nos *backends* testados, Unidade Central de Processamento (CPU) e GPU, também tiveram bons resultados.

6.2 Trabalho futuro

Um dos pontos em que pode ser desenvolvido trabalho no futuro é nas *features* que o modelo de Aprendizagem Automática utiliza. Atualmente o modelo utiliza uma *feature* para as leituras e outra para escritas feitas em cada *kernel*. Estas *features* são relativas a

leituras e escritas sobre a memória global. No entanto, existem outros tipos de operações deste tipo que não são contabilizadas como as operações sobre memória local.

Por outro lado, os modelos de Aprendizagem Automática utilizados para efetuar as previsões apenas são treinados uma vez e não sofrem mais alterações. Seria interessante que os modelos fossem adaptados à medida que são fornecidos mais dados.

Outro dos pontos que pode ser focado no futuro é a adaptação do trabalho desenvolvido à medida que são adicionados novos *backends* à *framework* Marrow.

Em relação à CPU, pode ser tida em consideração a carga da CPU quando se decide em que *backend* executar uma computação. Quando a carga da CPU é mais elevada pode ser vantajoso distribuir as computações por outros *backends*.

Por último, outro aspeto que pode ser explorado é o número de *threads* da CPU que devem ser utilizadas para minimizar o tempo de execução de uma computação. Pode ser utilizada uma rede de classificação, tal como é utilizada atualmente para o tamanho do bloco de *threads* da GPU.

Bibliografia

[1] M. Abadi et al. "TensorFlow: A System for Large-Scale Machine Learning". Em: *12th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2016, Savannah, GA, USA, November 2-4, 2016*. Ed. por K. Keeton e T. Roscoe. USENIX Association, 2016, pp. 265–283. URL: https://www.usenix.org/conference/osdi16/technical-sessions/presentation/abadi (ver p. 1).

[2] F. Agakov et al. "Using Machine Learning to Focus Iterative Optimization". Em: *Proceedings of the International Symposium on Code Generation and Optimization*. CGO '06. USA: IEEE Computer Society, abr. de 2006, pp. 295–305. ISBN: 0769524990. DOI: 10.1109/CGO.2006.37. URL: https://doi.org/10.1109/CGO.2006.37 (ver pp. 3, 13, 15, 19).

[3] F. Alexandre, R. Marqués e H. Paulino. "On the support of task-parallel algorithmic skeletons for multi-GPU computing". Em: *Symposium on Applied Computing, SAC 2014, Gyeongju, Republic of Korea - March 24 - 28, 2014*. Ed. por Y. Cho et al. ACM, 2014, pp. 880–885. DOI: 10.1145/2554850.2555018. URL: https://doi.org/10.1145/2554850.2555018 (ver p. 10).

[4] E. Alpaydin. *Introduction to Machine Learning*. 2nd. The MIT Press, fev. de 2010. ISBN: 9780262012430. DOI: 10.1017/S0269888910000056 (ver pp. 19, 20, 22–26).

[5] Avtech Scientific. *Advanced Simulation Library*. Versão 0.1.7. 9 de nov. de 2016. URL: http://asl.org.il/ (ver p. 1).

[6] T. Balaiah e R. Parthasarathi. "Autotuning of configuration for program execution in GPUs". Em: *Concurrency and Computation: Practice and Experience* (dez. de 2019). e5635 CPE-18-1587.R2, e5635. DOI: 10.1002/cpe.5635 (ver pp. 3, 13–16, 18).

[7] J. Bergstra, N. Pinto e D. Cox. "Machine learning for predictive auto-tuning with boosted regression trees". Em: *2012 Innovative Parallel Computing (InPar)*. Mai. de 2012, pp. 1–9. DOI: 10.1109/InPar.2012.6339587 (ver pp. 3, 13–15, 17).

[8] C. M. Bishop. *Pattern Recognition and Machine Learning (Information Science and Statistics)*. Berlin, Heidelberg: Springer-Verlag, 2006. ISBN: 0387310738 (ver pp. 22, 23).

[9] R. Chandra. *Parallel programming in OpenMP*. Morgan Kaufmann, 2001. ISBN: 1558606718. URL: https://www.worldcat.org/oclc/44883712 (ver p. 10).

[10] S. Cook. *CUDA programming: a developer's guide to parallel computing with GPUs*. Newnes, 2012 (ver p. 1).

[11] U. Dastgeer, J. Enmyren e C. Kessler. "Auto-tuning SkePU: A multi-backend skeleton programming framework for multi-GPU systems". Em: *Proceedings of the 4th International Workshop on Multicore Software Engineering*. New York, NY, USA: Association for Computing Machinery, mai. de 2011, pp. 25–32. DOI: 10.1145/1984693.1984697 (ver pp. 3, 13–18).

[12] T. L. Falch e A. C. Elster. "Machine Learning Based Auto-Tuning for Enhanced OpenCL Performance Portability". Em: *2015 IEEE International Parallel and Distributed Processing Symposium Workshop* (mai. de 2015). DOI: 10.1109/ipdpsw.2015.85. URL: http://dx.doi.org/10.1109/IPDPSW.2015.85 (ver pp. 3, 13–18).

[13] G. Fursin et al. "Milepost GCC: Machine Learning Enabled Self-tuning Compiler". Em: *Int. J. Parallel Program.* 39.3 (2011), pp. 296–327. DOI: 10.1007/s10766-010-0161-2. URL: https://doi.org/10.1007/s10766-010-0161-2 (ver pp. 3, 13, 15, 19).

[14] U. Garciarena e R. Santana. "Evolutionary Optimization of Compiler Flag Selection by Learning and Exploiting Flags Interactions:" em: *Proceedings of the 2016 on Genetic and Evolutionary Computation Conference Companion*. New York, NY, USA: Association for Computing Machinery, jul. de 2016, pp. 1159–1166. DOI: 10.1145/2908961.2931696 (ver pp. 3, 13–15, 19).

[15] R. Garg e L. Hendren. "A Portable and High-Performance General Matrix-Multiply (GEMM) Library for GPUs and Single-Chip CPU/GPU Systems". Em: *2014 22nd Euromicro International Conference on Parallel, Distributed, and Network-Based Processing*. Fev. de 2014, pp. 672–680. DOI: 10.1109/PDP.2014.40 (ver pp. 3, 13–16).

[16] S. Grauer-Gray et al. "Auto-tuning a high-level language targeted to GPU codes". Em: *2012 Innovative Parallel Computing (InPar)*. Mai. de 2012, pp. 1–10. DOI: 10.1109/InPar.2012.6339595 (ver pp. 3, 13–16).

[17] D. Grewe, Z. Wang e M. F. P. O'Boyle. "Portable mapping of data parallel programs to OpenCL for heterogeneous systems". Em: *Proceedings of the 2013 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. Fev. de 2013, pp. 1–10. DOI: 10.1109/CGO.2013.6494993 (ver pp. 3, 13–16).

[18] J. Guerreiro et al. "Multi-kernel Auto-Tuning on GPUs: Performance and Energy-Aware Optimization". Em: *2015 23rd Euromicro International Conference on Parallel, Distributed, and Network-Based Processing*. Mar. de 2015, pp. 438–445. DOI: 10.1109/PDP.2015.44 (ver pp. 3, 13–16).

[19] A. Hayashi et al. "Machine-learning-based performance heuristics for runtime cpu/gpu selection". Em: *Proceedings of the principles and practices of programming on the Java platform*. Set. de 2015, pp. 27–36 (ver pp. 3, 13–18).

[20] *HP Laptop 15-da3001TU*. https://store.hp.com/in-en/default/hp-laptop-pc-15-da3001tu-242d3pa.html, Acedido pela última vez a 2-12-2020 (ver p. 53).

[21] Q. Huang et al. "GPU as a General Purpose Computing Resource". Em: *2008 Ninth International Conference on Parallel and Distributed Computing, Applications and Technologies*. Los Alamitos, CA, USA: IEEE Computer Society, dez. de 2008, pp. 151–158. DOI: 10.1109/PDCAT.2008.38. URL: https://doi.ieeecomputersociety.org/10.1109/PDCAT.2008.38 (ver p. 1).

[22] D. P. Kingma e J. Ba. "Adam: A Method for Stochastic Optimization". Em: *3rd International Conference on Learning Representations, ICLR 2015, San Diego, CA, USA, May 7-9, 2015, Conference Track Proceedings*. Ed. por Y. Bengio e Y. LeCun. 2015. URL: http://arxiv.org/abs/1412.6980 (ver p. 45).

[23] C. Lancia et al. "Method to measure the mismatch between target and achieved received dose intensity of chemotherapy in cancer trials: a retrospective analysis of the MRC BO06 trial in osteosarcoma". Em: *BMJ Open* 9 (mai. de 2019), e022980. DOI: 10.1136/bmjopen-2018-022980 (ver p. 26).

[24] J. M. Lourenço. *The NOVAthesis LaTeX Template User's Manual*. NOVA University Lisbon. 2021. URL: https://github.com/joaomlourenco/novathesis/raw/master/template.pdf (ver p. v).

[25] C. Luk, S. Hong e H. Kim. "Qilin: Exploiting parallelism on heterogeneous multiprocessors with adaptive mapping". Em: *2009 42nd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. Dez. de 2009, pp. 45–55 (ver pp. 3, 13–16, 18).

[26] K. Ma et al. "GreenGPU: A Holistic Approach to Energy Efficiency in GPU-CPU Heterogeneous Architectures". Em: *41st International Conference on Parallel Processing, ICPP 2012, Pittsburgh, PA, USA, September 10-13, 2012*. IEEE Computer Society, 2012, pp. 48–57. DOI: 10.1109/ICPP.2012.31. URL: https://doi.org/10.1109/ICPP.2012.31 (ver pp. 3, 13–16).

[27] A. Magni, C. Dubach e M. F. P. O'Boyle. "Automatic optimization of thread-coarsening for graphics processors". Em: *International Conference on Parallel Architectures and Compilation, PACT '14, Edmonton, AB, Canada, August 24-27, 2014*. Ed. por J. N. Amaral e J. Torrellas. ACM, 2014, pp. 455–466. ISBN: 978-1-4503-2809-8. DOI: 10.1145/2628071.2628087. URL: https://doi.org/10.1145/2628071.2628087 (ver pp. 3, 13–15).

[28] A. Majumdar et al. "Dynamic GPGPU Power Management Using Adaptive Model Predictive Control". Em: *2017 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. Fev. de 2017, pp. 613–624. DOI: 10.1109/HPCA.2017.34 (ver pp. 3, 13–16).

[29] R. Marques et al. "Algorithmic Skeleton Framework for the Orchestration of GPU Computations". Em: *Euro-Par 2013 Parallel Processing - 19th International Conference, Aachen, Germany, August 26-30, 2013. Proceedings*. 2013, pp. 874–885. DOI: 10.1007/978-3-642-40047-6_86. URL: https://doi.org/10.1007/978-3-642-40047-6%5C_86 (ver pp. 3, 10).

[30] P. Martin et al. *OpenNN: Open Neural Network Library*. 2019. URL: https://www.opennn.net/ (ver p. 45).

[31] M. McMahon. *What Is a Computer Cluster?* https://www.easytechjunkie.com/what-is-a-computer-cluster.htm, Acedido pela última vez a 21-04-2021. Fev. de 2021 (ver p. 54).

[32] J. Michalakes e M. Vachharajani. "GPU Acceleration of Numerical Weather Prediction". Em: *Parallel Process. Lett.* 18.4 (2008), pp. 531–548. DOI: 10.1142/S0129626408003557. URL: https://doi.org/10.1142/S0129626408003557 (ver p. 1).

[33] A. Monsifrot, F. Bodin e R. Quiniou. "A Machine Learning Approach to Automatic Production of Compiler Heuristics". Em: *Artificial Intelligence: Methodology, Systems, and Applications, 10th International Conference, AIMSA 2002, Varna, Bulgaria, September 4-6, 2002, Proceedings*. Ed. por D. Scott. Vol. 2443. Lecture Notes in Computer Science. Springer, 2002, pp. 41–50. DOI: 10.1007/3-540-46148-5_5. URL: https://doi.org/10.1007/3-540-46148-5%5C_5 (ver pp. 3, 13, 15, 16, 19).

[34] A. Munshi. "The OpenCL specification". Em: *2009 IEEE Hot Chips 21 Symposium (HCS)*. 2009, pp. 1–314. DOI: 10.1109/HOTCHIPS.2009.7478342 (ver pp. 1, 10).

[35] *NN-SVG*. http://alexlenail.me/NN-SVG/index.html, Acedido pela última vez a 25-11-2020 (ver p. 44).

[36] NVIDIA. *CUDA C++ Best Practices Guide*. https://docs.nvidia.com/cuda/cuda-c-best-practices-guide/index.html, Acedido pela última vez a 03-11-2021. 2021 (ver p. 8).

[37] NVIDIA. *NVIDIA TESLA V100 GPU ARCHITECTURE*. http://images.nvidia.com/content/volta-architecture/pdf/volta-architecture-whitepaper.pdf, Acedido pela última vez a 04-02-2020. 2017 (ver pp. 7, 8).

[38] *Online K Means Clustering Calculator*. https://people.revoledu.com/kardi/tutorial/kMean/Online-K-Means-Clustering.html, Acedido pela última vez a 17-03-2021 (ver pp. 31, 32).

[39] W. C. Skamarock et al. "A Description of the Advanced Research WRF Version 2". Em: jun. de 2005 (ver p. 1).

[40] F. Soldado, F. Alexandre e H. Paulino. "Execution of Compound Multi-Kernel OpenCL Computations in Multi-CPU/Multi-GPU Environments". Em: *Concurrency and Computation: Practice & Experience* 28.3 (mar. de 2016), pp. 768–787. ISSN: 1532-0626. DOI: 10.1002/cpe.3612. URL: https://doi.org/10.1002/cpe.3612 (ver pp. 3, 13–16).

[41] T. Srivastava. *Introduction to k-Nearest Neighbors: A powerful Machine Learning Algorithm (with implementation in Python 'I&' R)*. https://www.analyticsvidhya.com/blog/2018/03/introduction-k-neighbours-algorithm-clustering/, Acedido pela última vez a 11-01-2020. 2018 (ver p. 21).

[42] M. Stephenson e S. Amarasinghe. "Predicting Unroll Factors Using Supervised Classification". Em: *Proceedings of the International Symposium on Code Generation and Optimization*. CGO '05. USA: IEEE Computer Society, 2005, pp. 123–134. ISBN: 076952298X. DOI: 10.1109/CGO.2005.29. URL: https://doi.org/10.1109/CGO.2005.29 (ver pp. 3, 13, 15, 16, 19).

[43] *Technical Description*. https://cluster.di.fct.unl.pt/docs/technical/, Acedido pela última vez a 21-04-2021. Ago. de 2020 (ver p. 54).

[44] P. Tillet e D. Cox. "Input-Aware Auto-Tuning of Compute-Bound HPC Kernels". Em: *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. Nov. de 2017 (ver pp. 3, 13–15).

[45] Z. Wang e M. F. O'Boyle. "Mapping Parallelism to Multi-cores: A Machine Learning Based Approach". Em: *Proceedings of the 14th ACM SIGPLAN symposium on Principles and practice of parallel programming*. Vol. 44. Abr. de 2009, pp. 75–84 (ver pp. 3, 13–16, 19).

[46] Y. Wen, Z. Wang e M. F. P. O'Boyle. "Smart multi-task scheduling for OpenCL programs on CPU/GPU heterogeneous platforms". Em: *2014 21st International Conference on High Performance Computing (HiPC)*. Dez. de 2014, pp. 1–10. DOI: 10.1109/HiPC.2014.7116910 (ver pp. 3, 13–18).

[47] B. van Werkhoven. "Kernel Tuner: A search-optimizing GPU code auto-tuner". Em: *Future Generation Computer Systems* 90 (2019), pp. 347–358. DOI: 10.1016/j.future.2018.08.004. URL: https://doi.org/10.1016/j.future.2018.08.004 (ver pp. 3, 14).

[48] Yixun Liu, E. Z. Zhang e X. Shen. "A cross-input adaptive framework for GPU program optimizations". Em: *2009 IEEE International Symposium on Parallel Distributed Processing*. Mai. de 2009, pp. 1–10. DOI: 10.1109/IPDPS.2009.5160988 (ver pp. 3, 13–16, 18).

[49] C. Yu e S. Tsao. "Efficient and Portable Workgroup Size Tuning". Em: *IEEE Transactions on Parallel and Distributed Systems* 31.2 (fev. de 2020), pp. 455–469. ISSN: 2161-9883. DOI: 10.1109/TPDS.2019.2937295 (ver pp. 3, 13–16).