UNIVERSIDADE FEDERAL DE SANTA MARIA
CENTRO DE TECNOLOGIA
CIÊNCIA DA COMPUTAÇÃO

AUMENTO DO DESEMPENHO DA SUPPORT VECTOR MACHINE ATRAVÉS DE TÉCNICAS DE PARALELIZAÇÃO DE CÓDIGO EM GPU

PROJETO DE TRABALHO DE GRADUAÇÃO

Henry Cagnini

Santa Maria, RS, Brasil

2014

AUMENTO DO DESEMPENHO DA SUPPORT VECTOR MACHINE ATRAVÉS DE TÉCNICAS DE PARALELIZAÇÃO DE CÓDIGO EM GPU

Henry Cagnini

Projeto de Trabalho de Graduação a ser apresentado ao curso de Ciência da Computação da Universidade Federal de Santa Maria (UFSM, RS), como requisito parcial para a obtenção do grau de

Bacharel em Ciência da Computação

Orientador: Prof.ª Dr.ª Ana T. Winck

Trabalho de Graduação Nº 374
Santa Maria, RS, Brasil

2014

Universidade Federal de Santa Maria
Centro de Tecnologia
Ciência da Computação

A comissão Examinadora, abaixo assinada,
aprova

AUMENTO DO DESEMPENHO DA SUPPORT VECTOR MACHINE ATRAVÉS DE TÉCNICAS DE PARALELIZAÇÃO DE CÓDIGO EM GPU

elaborado por **Henry Emanuel Leal Cagnini**

como requisito para obtenção do grau de
Bacharel em Ciência da Computação

Comissão Examinadora

Ana Trindade Winck, Drª.
(Presidente/Orientador)

Andrea Schwertner Charão, Drª.
(UFSM)

Mateus Beck Rutzig, Drº.
(UFSM)

Santa Maria, 9 de julho de 2014.

AGRADECIMENTOS

Agradeço primeiramente aos meus pais, Marlene Leal Cagnini e Luiz Sadi Cagnini, por terem me dado a educação e motivação que me permitiram ser o que sou hoje;

a minha namorada Juliana de Oliveira Mozzaquatro, por ter me apoiado nos momentos difíceis desde sempre;

a Profª Ana Trindade Winck, por ter me acolhido como seu orientando e dado a orientação necessária a realização desse trabalho;

e finalmente a Universidade Federal de Santa Maria, por ser um ambiente acadêmico propício ao desenvolvimento intelectual de seus alunos.

RESUMO

Tese de Conclusão de Curso
Curso de Ciência da Computação
Universidade Federal de Santa Maria

Aumento do desempenho da Support Vector Machine através de técnicas de paralelização de código em GPU

AUTOR: Henry Emanuel Leal Cagnini
ORIENTADOR: Ana Trindade Winck

Data e Local da Defesa: Santa Maria, 9 de julho de 2014.

A Support Vector Machine é um método de classificação de dados que vem experimentando um aumento na sua popularidade. Este aumento se deve principalmente à publicação da LIBSVM, biblioteca de código fonte que a implementa e está disponível em diversas linguagens de programação. Desde sua publicação, vários trabalhos vêm buscando otimizar seu desempenho através da paralelização de trechos de seu código fonte, sobretudo com a utilização da *Graphics Processor Unit*, a GPU. Alguns destes trabalhos fazem uso do *framework* CUDA, ficando restritos a GPUs da fabricante NVIDIA. Este trabalho propõe a paralelização da Support Vector Machine com a utilização do *framwork* OpenCL, permitindo que a solução seja portável para diferentes GPUs. A fase do processo classificatório escolhida para a otimização foi a de treino. A solução final obteve *speedup* de três vezes em relação a versão sequencial da LIBSVM, e foi desenvolvida utilizando duas GPUs de arquiteturas distintas.

Palavras chave: Programação paralela em GPU, OpenCL, Support Vector Machine.

ABSTRACT

Course Conclusion Thesis
Computer Science Course
Federal University of Santa Maria

Enhancing the Support Vector Machine performance through techinics of code parallelization in GPU

AUTHOR: Henry Emanuel Leal Cagnini
ADVISOR: Ana Trindade Winck
Defense place and Date: Santa Maria, july 9th 2014.

The Support Vector Machine is a data classification method that has presented an increase in terms of its popularity. This increase is mainly due to the publication of LIBSVM, a library that implements Support Vector Machine which source code is available in a diversity of programming languages. Since its publication, several researches have tried to optimize its performance through parallelization of parts of its source code, mostly using the Graphics Processor Unit as a parallel processing device. Some of these work use the CUDA framework, limiting the solutions to the NVIDIA's GPUs. This work proposes the parallelization of the Support Vector Machine through OpenCL framework, allowing the solution to be executed in a wider variety of GPUs. We choose training phase of the classificatiom process for the optimization. The final solution achieved a speedup of three in comparisson to the sequential version, and it is developed using two distinct GPU architectures.

Keywords: GPU parallel programming, OpenCL, Support Vector Machne.

SUMÁRIO

1 INTRODUÇÃO

A mudança da plataforma analógica para digital no armazenamento de dados permitiu que o custo do dado armazenado decaísse. Tal barateamento fez com que cada vez mais dados fossem sendo estocados.

Em decorrência do grande volume de dados que agora podem ser armazenados, surge o impasse de como transformar um número ou um nome em conhecimento, algo com finalidade e uso. Para isto, existem os algoritmos de aprendizado de máquina, capazes de operar com ou sem supervisão humana.

Um destes algoritmos é a *Support Vector Machine* (SVM, na sigla em inglês), introduzida em (VAPNIK; CORTES, 1995). A SVM vem experimentando um aumento na sua popularidade, em parte devido a publicação da LIBSVM (CHANG; LIN, 2011), uma implementação da SVM disponível em diversas linguagens de programação.

O diferencial da SVM em relação a outros métodos consiste em abordar os dados de maneira estatística, situando-os em um espaço cartesiano multidimensional onde a distância euclidiana entre eles determina seu grau de correlação. Seu uso expandiu-se a ponto de ser utilizada em programas de análise de dados, como o Weka (WITTEN; FRANK; HALL, 2011).

1.1 Motivação

De um algoritmo de aprendizado de máquina como a SVM se espera, além de que possa realizar a tarefa de classificação satisfatoriamente, tratar novos problemas de disponibilidade de dados como o *Big Data* (WITTEN; FRANK; HALL, 2011). A abordagem dos novos desafios envolve recorrer a técnicas de computação mais eficientes, sendo o processamento paralelo uma delas.

Uma das primeiras abordagens de processamento paralelo surgiu com as redes de computadores (FOSTER, 1995). Através da interconexão de diversas máquinas operando sob um mesmo protocolo de comunicação é possível o envio, processamento remoto e retorno de informações. Esta abordagem de paralelismo evoluiu de forma a abranger outros conceitos, e é agora denominada como computação distribuída.

Apesar de a computação distribuída ser uma opção viável à implementação da SVM, algumas questões surgem, como latência de resposta, segurança de dados, heterogeneidade de máquinas, dentre outras. Tais conceitos exigem um tratamento específico para contorná-los,

limitando o ganho de uma SVM paralelizada em uma rede distribuída.

Em contrapartida, o surgimento da *General Purpose Graphics Processor Unit*, GPGPU ou apenas GPU (TECHSPOT, 2014) permite outra abordagem de programação paralela, mais adequada a aplicações com tempo de execução na casa das horas. Sendo a GPU um *hardware* comum à maioria dos computadores domésticos e com um potencial pouco utilizado fora de aplicações gráficas intensas, torna-se atrativo analisá-la e explorar os ganhos que ela possa oferecer a um algoritmo de aprendizado de máquina.

1.2 Objetivos

Propõe-se aumentar o desempenho geral da SVM enquanto utilizada para tarefa de classificação de dados através da paralelização de seu código fonte: uma parcela será executada na GPU, enquanto o restante será executado em sua concepção original na CPU. Para que um maior número de arquiteturas de GPU possam usufruir dos benefícios da SVM paralelizada, optou-se por utilizar o *framework* OpenCL para escrita do código fonte paralelo. Espera-se, ainda, manter os mesmos níveis de acurácia dos modelos preditivos gerados em relação a uma SVM não paralelizada.

2 REVISÃO DA LITERATURA

Mesmo sendo possível apenas aplicar técnicas de otimização de código fonte à SVM e obter um desempenho satisfatório, a qualidade do resultado final poderia ser contestável por não possuir nenhum respaldo teórico. É preciso compreender tanto o funcionamento da SVM quanto das ferramentas, equipamentos e técnicas que se propõem a aumentar seu desempenho. Assim, é possível explorar as particularidades de cada componente envolvido na otimização e extrair o melhor que cada um tem a oferecer.

2.1 Graphics Processor Unit

A *Graphics Processor Unit* (GPU, na sigla em inglês) é um *hardware* concebido principalmente para a renderização de gráficos. As GPUs modernas datam de 1995, quando adquiriram seu formato modular aos computadores pessoais (TECHSPOT, 2014). Os dispositivos anteriores a esta data dificilmente se encaixariam no conceito das placas de vídeo modernas, por possuírem uma arquitetura e programação dependentes do hardware com o qual se comunicavam. Portanto, quando o termo GPU for utilizado, será referida a GPU contemporânea, mais precisamente a *General Purpose Graphics Processor Unit* (ou GPGPU).

A GPU é caracterizada por conter vários processadores paralelos designados a desempenhar tarefas simples, que não possuam controle de fluxo intenso em seu código fonte (CATANZARO; SUNDARAM; KEUTZER, 2008). Os processadores são ainda especializados em diferentes etapas do processo de renderização de vídeo, fazendo com que a GPU seja um dispositivo com alto grau de paralelismo tanto em número de processadores quanto em tarefas desempenhadas simultaneamente (HARDING; BANZHAF, 2007).

O fato de que cada processador é capaz de realizar várias operações de ponto flutuante por *clock* da placa de vídeo (HARDING, BANZHAF, 2007) despertou o interesse de utilizá-la para aplicações que não apenas a renderização. O desafio de explorar outras possibilidades de processamento advém do fato da GPU interpretar apenas comandos relativos à renderização, requirindo que o código fonte de programas não gráficos sejam traduzidos para comandos gráficos. Os primeiros esforços neste sentido foram dados pela NVIDIA em 2006, resultando no *framework* CUDA - *Compute Unified Device Architecture* (NVIDIA, 2014a). O CUDA é específico para placas gráficas da NVIDIA, e permite escrever programas que são executados parcialmente na GPU. Posteriormente, (KHRONOS, 2012) lança o OpenCL, *framework* com

mesmo propósito e maior número de fabricantes de GPU suportados.

2.1.1 Motivação

Tanto a GPU quanto uma CPU com mais de um *core* (núcleo, ou processador paralelo) podem executar aplicações paralelizadas. A vantagem da GPU sobre a CPU é seu número muito superior de processadores paralelos. As CPUs modernas possuem de dois a oito processadores (de fato, ambas as CPUs utilizadas neste trabalho possuem quatro, como pode ser visualizado na Tabela 3.1 da seção 3.1). Em contrapartida, na Tabela 3.2 da mesma seção 3.1, no campo de *stream processors*, a GPU mais modesta utilizada possui 240 processadores paralelos, ou 60× mais processadores que as CPUs usadas. Desde que respeite-se sua limitação computacional (referente ao desempenho inferior à CPU quando processando controle de fluxo intenso), a GPU pode extrair um paralelismo superior à CPU que, dependendo da aplicação, pode significar um aumento substancial de desempenho.

2.2 OpenCL

O OpenCL é um *framework* que prioriza a portabilidade, permitindo que um código fonte possa ser executado em arquiteturas de *hardware* distintas da CPU, como a GPU. Uma de suas principais funções é abstrair a visão do conjunto CPU, memória principal e outros dispositivos suportados para uma visão que possa ser interpretada e manipulada pelo programador. A visão abstraída é interpretada de diversas formas, que variam da configuração física ao modo como o código fonte é executado.

2.2.1 Plataforma

Segundo (KHRONOS, 2012), a configuração lógica de uma máquina com OpenCL é dividida em dois módulos: hospedeiro, composto de memória principal e CPU, e dispositivo suportado, sendo este um dispositivo capaz de executar funções kernel (OpenCL). Apesar de existir mais de um tipo arquitetura capaz da execução do kernel (OpenCL), neste trabalho é considerado apenas a GPU, dispositivo sobre o qual a paralelização da *Support Vector Machine* ocorrerá. A GPU possui, além de seus processadores paralelos, uma memória dedicada que pode ser acessada através do *framework*. A velocidade de leitura e escrita da memória dedicada é geralmente muito superior à velocidade apresentada pela memória principal do hospedeiro

(CATANZARO; SUNDARAM; KEUTZER, 2008), porém a latência elevada de acesso do hospedeiro faz com que ela seja mais utilizada pela própria GPU.

2.2.2 Programação e Execução

A programação e execução de um programa é análoga à divisão lógica do OpenCL, sendo um trecho do código fonte executado no hospedeiro e outro na GPU. A parte executada no hospedeiro pode ser escrita em qualquer linguagem suportada pelo *framework* (como C++ ou Javascript), e seu espaço de memória é endereçado na memória principal. É também no hospedeiro que a invocação do código fonte da GPU é feita, podendo posicioná-la em qualquer etapa da execução do programa. O código fonte da GPU, por outro lado, deve ser escrito em linguagem C, mais especificamente o padrão C99, e é constituído por dois tipos de funções: kernel (OpenCL) e interna.

Uma função interna não possui comunicação com o meio externo da GPU, e seu propósito é tornar o código fonte mais legível ao programador. Funções kernel (OpenCL) são diretamente invocadas a partir do código fonte hospedeiro, são obrigatoriamente do tipo void (sem retorno) e escrevem a saída do procesamento em uma variável ou vetor provido como parâmetro para a função. Uma função kernel (OpenCL) é invocada várias vezes por diversos processadores (também chamados de *threads* da GPU) que podem ou não comunicar-se durante a execução. Caso existam mais invocações do que processadores disponíveis, essas disputarão tempo de GPU para execução no processador.

2.2.3 Portabilidade e desempenho

Pelo fato de o OpenCL ser um *framework* portável, compatível com uma diversidade de *hardware*, seu desempenho tende a ser pior que o CUDA, específico para GPUs da NVIDIA. Segundo (KARIMI; DICKSON; HAMZE, 2010), a taxa de transferência da memória principal para a GPU é superior quando CUDA é utilizado, além dele possuir uma curva de aprendizado mais amigável. Contudo, esse limitante no desempenho não deve ser considerado como um fator de exclusão do OpenCL de trabalhos de otimização, pois seu desempenho é constantemente inferior ao de CUDA (KARIMI; DICKSON; HAMZE, 2010); apenas um desempenho decrescente de acordo com a complexidade da aplicação - isto é, aplicações com controle de fluxo mais intenso - tornaria-o pouco atraente para trabalhos de otimização.

2.3 Support Vector Machine

A *Support Vector Machine*, mais conhecida pela sigla SVM, é um método de aprendizado de máquina introduzida em (VAPNIK; CORTES, 1995) e embasada na Teoria do Aprendizado Estatístico (VAPNIK, 1999). Sua abordagem é estatística sobre os dados de entrada, os *datasets*, situando-os em um espaço cartesiano onde o processo classificatório ocorre.

2.3.1 Motivação

Existem diversos métodos de classificação de dados, mas a *Support Vector Machine* vem experimentando um aumento na sua popularidade ultimamente. Isto deve-se ao fato de ser um algoritmo de classificação com diversos recursos computacionais que maximizam a generalização do problema, e relativamente mais recente que outras abordagens de classificação. Por capacidade de generalização entende-se a miniminização da possibilidade de classificar erroneamente dados que não foram utilizados na fase de treinamento (DE SOUZA, 2005).

É possível evidenciar esse fenômeno comparando o volume de publicações contendo o nome dos métodos de classificação disponíveis em uma popular ferramenta de análise de dados, o Weka (WITTEN; FRANK; HALL, 2011). O nome dos classificadores foram adaptados, e são listados na primeira coluna das Tabelas 2.1 (classificadores baseados em Bayes), 2.2 (baseados em árvores), 2.3 (regras), 2.4 (funções), 2.5 (abordagem adiada), 2.6 (múltipla instância), e os que não se enquadram nas categorias anteriores na Tabela 2.7. Os métodos foram pesquisados na base de conhecimento Scopus (ELSEVIER, 2014) na área de "ciências físicas", sem limite de data de publicação, juntamente aos termos descritos nas duas últimas colunas. Os termos foram pesquisados nas áreas de título, resumo e palavras-chave dos documentos da base.

O volume de publicações contendo o termo *Support Vector Machine* é maior tanto para documentos voltados a classificação quanto para *data mining*. Na Figura 2.1 é possível visualizar o número de publicações contendo o termo "*Support Vector Machine*" na área "ciências físicas", no decorrer dos anos de 1990 e 2013.

A Scopus não permite pesquisar palavras-chave no texto de seus documentos, não sendo viável afirmar com precisão o impacto da LIBSVM no interesse pela *Support Vector Machine*. Porém, o número de documentos publicados entre 2011 (ano da publicação da LIBSVM) e 2012 cresce mais acentuadamente do que o interesse visto de 2009 em diante, período em que o número de publicações parecia se estabilizar. Com ou sem influência da LIBSVM, porém, a

Método	classification	data mining
Averaged one dependence estimator	30	12
Bayesian Logistic Regression	195	47
Bayes Net	92	39
Complement Naïve Bayes	15	4
Discriminative Multinomial Naïve Bayes	7	1
Hidden Naïve Bayes	63	30
Naïve Bayes	2.624	776
Naïve Bayes Multinomial	87	13
Naïve Bayes Multinomial Updateable	0	0
Simple Naïve Bayes	292	90
Updateable Naïve Bayes	0	0
Weightily averaged one dependence estimator	2	1

Tabela 2.1 – Métodos de classificação baseados em Bayes da ferramenta Weka, pesquisados na base de conhecimento Scopus junto com os termos das segunda e terceira colunas. Pesquisa realizada em 2 de julho de 2014.

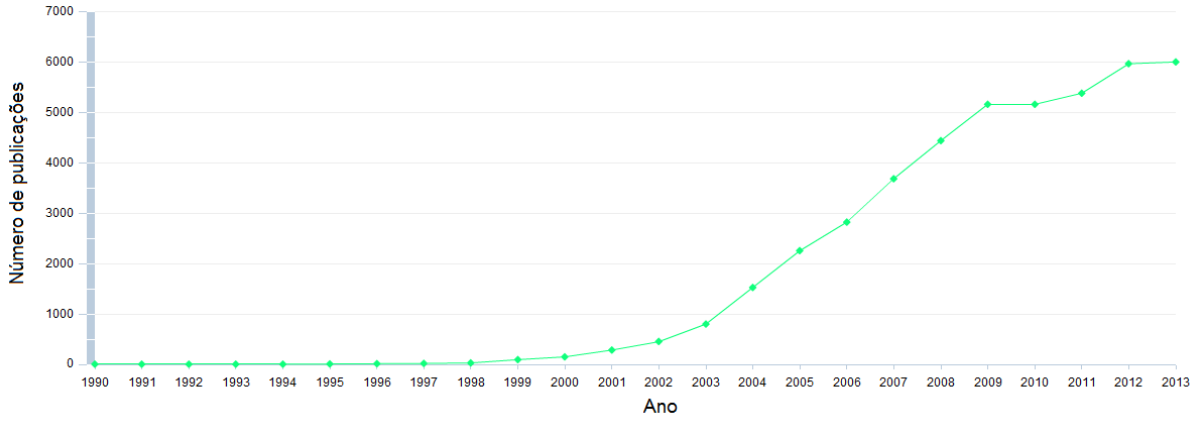

Figura 2.1 – Volume de publicações *versus* ano contendo o termo "*Support Vector Machine*" na área "ciências físicas", da base de conhecimento Scopus.

Support Vector Machine mostra-se um método de aprendizado de máquina popular, merecendo interesse tanto como ferramenta de classificação como objeto de estudo.

2.3.2 Dataset

Um *dataset* (conjunto de dados em português) é um conjunto de informações sobre as quais deseja-se chegar a uma conclusão e podem, mas não necessariamente são, representados graficamente por uma tabela. As informações descritas no *dataset* são das mais diversas naturezas: uma pesquisa de opinião de eleitores, dados sensoriais de um sistema de previsão de tempo, relatório anual de lucro de uma empresa, dentre outras possibilidades. Para propósito de esclarecimento, utilizaremos o dataset descrito na Tabela 2.8, traduzido de (WITTEN; FRANK;

Método	classification	data mining
Alternating Decision Tree	25	10
Best First Decision Tree	392	190
One Level Decision Tree	237	103
Functional Tree	542	1.239
ID3	372	266
J48	238	137
J48 graft	2	2
LAD Tree	3	2
Logistic Model Tree	427	188
M5P	2	10
Naïve Bayes Tree	677	268
Random Forest	2.005	398
Random Tree	2.045	522
Reduced Error Pruning Tree	8	6
Simple CART Tree	72	22

Tabela 2.2 – Métodos de classificação baseados em árvores da ferramenta Weka, pesquisados na base de conhecimento Scopus junto com os termos das segunda e terceira colunas. Pesquisa realizada em 2 de julho de 2014.

Método	classification	data mining
Conjuctive Rule	1	2
Decision Table	731	623
Decision Table Naïve Bayes	26	15
Ripper	110	71
M5 Rules	1	9
Nearest Neighbor	7.497	1.203
OneR	25	10
Partial decision tree	133	60
Prism	241	55
Ridor	13	10
ZeroR	3	1

Tabela 2.3 – Métodos de classificação baseados em regras da ferramenta Weka, pesquisados na base de conhecimento Scopus junto com os termos das segunda e terceira colunas. Pesquisa realizada em 2 de julho de 2014.

Método	classification	data mining
Gaussian Process	1.633	194
Isotonic Regression	19	2
LesatMedSq	0	0
LibLinear	19	1
Support Vector Machine	22.045	2.610
Linear Regression	3.109	920
Logistic Regression	2.306	585
Multilayer Perceptron	2.248	207
Pace Regression	11	9
Least square direction	107	20
Radial Basis Function Network	2.833	258
Sequential Minimal Optimization	170	31
Stochastic Gradient Descent	99	22
Voted Perceptron	9	4
Mistake Driven Perceptron	1	1

Tabela 2.4 – Métodos de classificação baseados em funções da ferramenta Weka, pesquisados na base de conhecimento Scopus junto com os termos das segunda e terceira colunas. Pesquisa realizada em 2 de julho de 2014.

Método	classification	data mining
Basic Nearest Neighbor	216	45
k Nearest Neighbor	4.572	695
Lazy Bayesian Rule	10	4
Locally Weighted Learning	55	14

Tabela 2.5 – Métodos de classificação baseados em abordagem adiada da ferramenta Weka, pesquisados na base de conhecimento Scopus junto com os termos das segunda e terceira colunas. Pesquisa realizada em 2 de julho de 2014.

Método	classification	data mining
Citation KNN	11	0
Diverse Density	192	40
Multiple Instance Boost	8	5
EM Diverse Density	3	0
Multi Instance Regression	63	13
Multi Instance Logistic Regression	15	2
Nearest Neighbor Kullback Leibler	28	2
Voting feature interval	20	2

Tabela 2.6 – Métodos de classificação baseados em múltipla instância da ferramenta Weka, pesquisados na base de conhecimento Scopus junto com os termos das segunda e terceira colunas. Pesquisa realizada em 2 de julho de 2014.

Método	classification	data mining
Hyper Pipe	5	1
Voting feature Intervals	20	2

Tabela 2.7 – Métodos de classificação da ferramenta Weka que não se enquadram nas abordagens anteriores, pesquisados na base de conhecimento Scopus junto com os termos das segunda e terceira colunas. Pesquisa realizada em 2 de julho de 2014.

HALL, 2011).

Aparência	Temperatura	Umidade	Vento	Jogável
Ensolarado	Quente	Alta	Não	Não
Ensolarado	Quente	Alta	Sim	Não
Nublado	Quente	Alta	Não	Sim
Chuvoso	Ameno	Alta	Não	Sim
Chuvoso	Frio	Normal	Não	Sim
Chuvoso	Frio	Normal	Sim	Não
Nublado	Frio	Normal	Sim	Sim
Ensolarado	Ameno	Alta	Não	Não
Ensolarado	Frio	Normal	Não	Sim
Chuvoso	Ameno	Normal	Não	Sim
Ensolarado	Ameno	Normal	Sim	Sim
Nublado	Ameno	Alta	Sim	Sim
Nublado	Quente	Normal	Não	Sim
Chuvoso	Ameno	Alta	Sim	Não

Tabela 2.8 – Um *dataset* com informações relativas a condições climáticas propícias ou não para a prática de uma atividade não especificada. Traduzido de (WITTEN; FRANK; HALL, 2011).

O dataset da Tabela 2.8 possui 14 registros, também denominados dados, sendo cada registro uma linha, e 5 atributos: aparência, temperatura, umidade, vento e jogável, ocupando as colunas. Os quatro primeiro atributos são denominados atributos preditivos, e o último (jogável) se chama atributo classe. O valor do atributo classe é a conclusão sobre a qual deseja-se chegar a partir dos valores dos atributos preditivos do *dataset*.

Os atributos preditivos de um *dataset* podem se relacionar com o atributo classe de três formas: sendo fortemente relacionados (quando a mudança do valor de um atributo preditivo impacta diretamente no valor do atributo classe), fracamente relacionados (quando o valor de um atributo preditivo depende do valor de outros atributos para resultar em uma variação do atributo classe), e não relacionados. É comum em tarefas de mineração de dados pré-processar os *datasets*, de forma a eliminar os atributos preditivos menos relevantes na constituição do valor do atributo classe.

O pré-processamento trabalha também com outras questões que não apenas a eliminação

de atributos preditivos irrelevantes, mas também como, por exemplo, dados ruidosos. Um dado é considerado um ruído quando o valor de um atributo preditivo não segue a tendência dos outros dados para o mesmo valor de atributo preditivo. Na Tabela 2.8, um dado seria um ruído caso apresentasse um valor de Umidade negativo, pois seria destoante do intervalo de valores previamente apresentados por este atributo no *dataset*.

Com base nos valores de atributos preditivos de um dado *dataset* que se saiba a classe de todos os registros, e desde que o valor da classe seja discreto (sendo um rótulo, e não um número, como na Tabela 2.8), é possível gerar um modelo preditivo através do processo de classificação, que permitirá determinar a classe de novos registros que venham a ser inseridos no *dataset*.

2.3.3 Classificação de dados

A classificação é um processo desempenhado por um algoritmo que visa classificar dados sobre os quais não se sabe o valor de seu atributo classe. Existem diversos algoritmos que podem realizar esse processo, cada um com uma abordagem particular. Neste trabalho é descrito o funcionamento da *Support Vector Machine* (VAPNIK, 1999).

A SVM representa um *dataset* de forma matemática, fazendo um mapeamento de suas características para um espaço cartesiano. Cada dado do *dataset* é transformado em um ponto, cada atributo em um eixo e o número de atributos do *dataset* determina a dimensão do espaço cartesiano. No *dataset* da Tabela 2.8, o espaço cartesiano tem dimensão R^5 e é povoado por 14 pontos. Na Figura 2.2 uma simplificação do espaço cartesiano é demonstrada com 3 dimensões, sendo os atributos umidade, aparência e jogável os escolhidos para localização dos pontos.

O objetivo da SVM, então, é traçar um hiperplano de forma que os dados sejam separados linearmente. É perceptível que no exemplo da Figura 2.2 não é possível fazer tal separação, pois nenhuma reta que seja traçada separará os dados das duas classes de forma linear, colocando uma classe para um lado e outra para o outro do hiperplano. Esta impossibilidade ocorre quando os dados são visualizados em uma dimensão muito baixa. Para que seja possível fazer a separação, a SVM realiza um mapeamento dos dados do espaço de entrada para um espaço de características (*feature space*) com dimensão suficiente alta para a separação. O espaço de características do *dataset* da Tabela 2.8 é demonstrado na Figura 2.3.

Uma vez no espaço de características, inicia-se a tarefa de desenho do hiperplano. Para isto, *support vectors* são delimitados, e o hiperplano é escolhido de forma a manter-se o mais

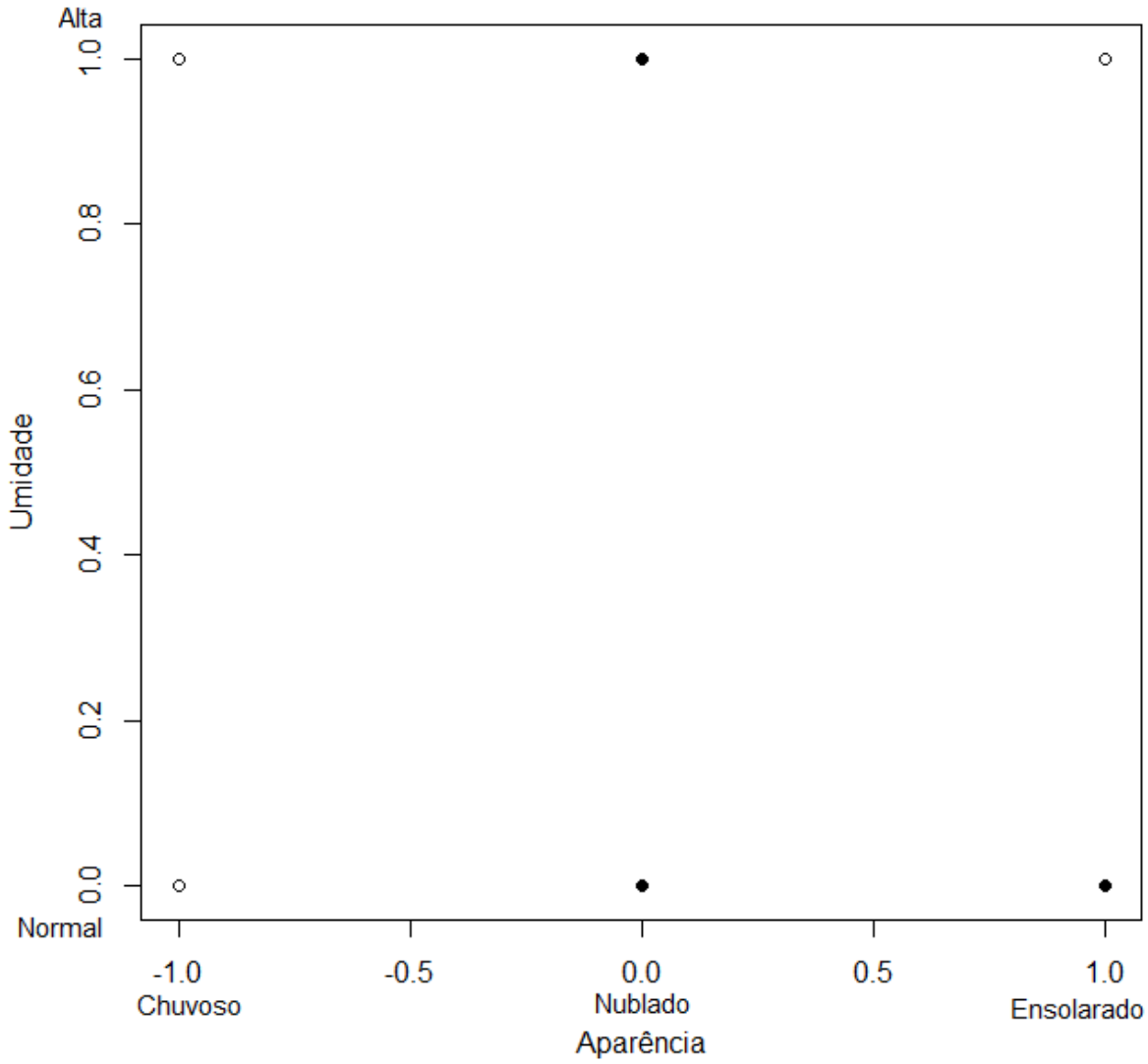

Figura 2.2 – Espaço de entrada do *dataset* da Tabela 2.8, demonstrando as seis possíveis combinações dos atributos aparência, umidade e jogável. Os pontos pretos possuem o valor jogável igual a "sim", enquanto os pontos brancos possuem jogável igual a "não".

distante possível das margens dos *support vectors*. Um *support vector* é uma reta constituída por dados de uma classe que se encontram mais próximos dos dados de outra classe. A maximização da distância euclidiana dos *support vectors* é necessária para se evitar que dados futuros sejam erroneamente classificados. Na Figura 2.4, retirada de (WITTEN; FRANK; HALL, 2011), é possível visualizar um hiperplano contido entre dois *support vectors*.

O hiperplano gerado pelo processo de maximização é um modelo preditivo, que pode posteriormente ser utilizado para classificar dados novos ao *dataset*. É comum verificar a acurácia do modelo gerado através de técnicas de validação ou do uso de dados sobre os quais se sabe o valor do atributo classe, mas que não foram utilizados na geração do modelo.

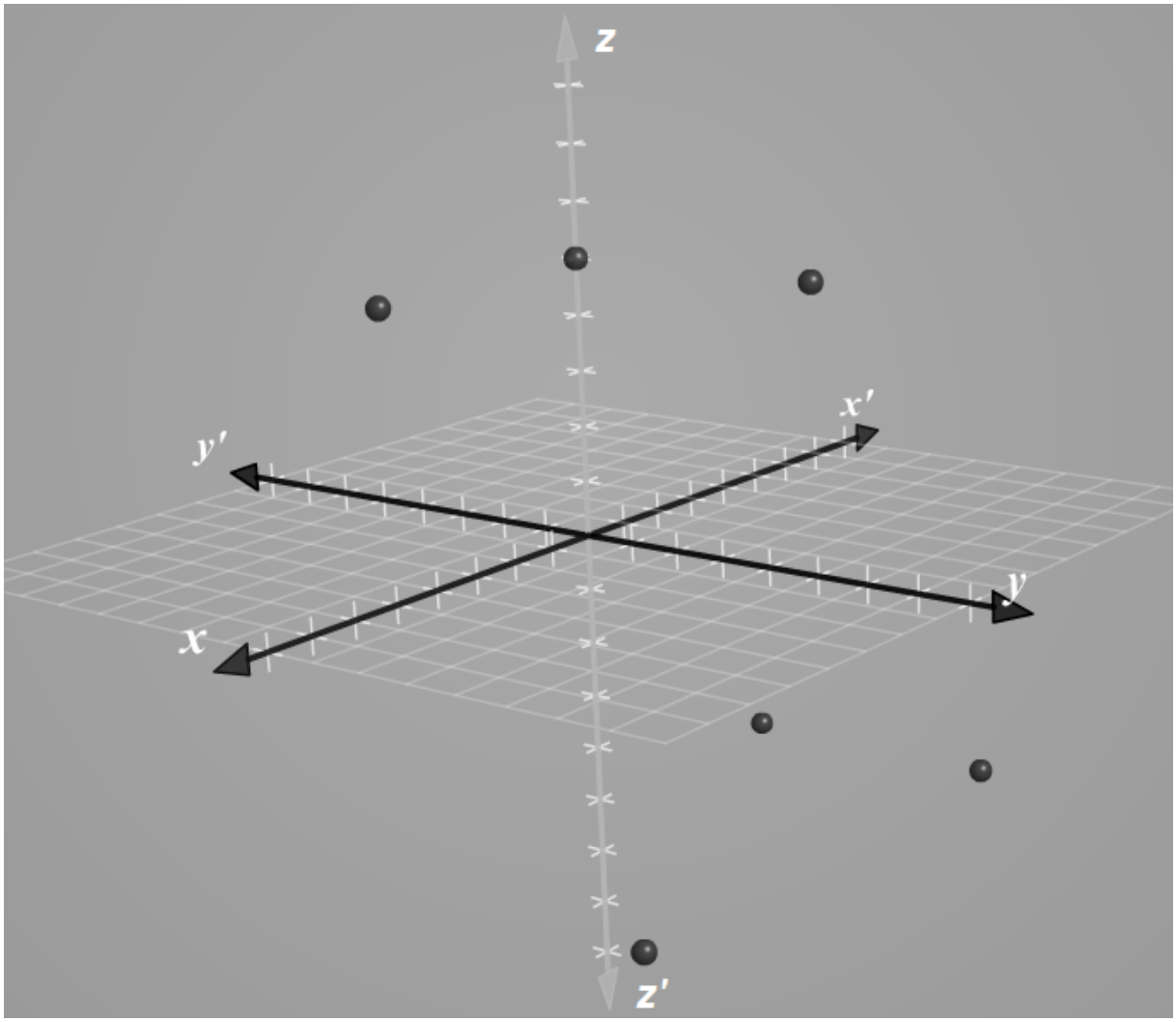

Figura 2.3 – Espaço de características do *dataset* da Tabela 2.8. Os pontos acima do plano xy possuem a classe jogável igual a "sim", e os pontos abaixo possuem classe igual a "não".

2.4 Trabalhos relacionados

Já foram realizados trabalhos que otimizam a SVM através da paralelização. As abordagens diferenciam-se pela etapa da classificação otimizada e ferramentas utilizadas. Na Tabela 2.9 é possível ver uma síntese das características de cada trabalho, incluindo o que se apresenta neste documento. No decorrer desta seção serão explicados em detalhes onde cada modificação atua para melhorar o desempenho da SVM.

Em (LU; ROYCHOWDHURY; VANDENBERGHE, 2008), os *support vectors* são compartilhados entre diversas máquinas, conectadas em uma rede de computadores. O objetivo desta abordagem é distribuir o trabalho de maximização das margens do hiperplano em relação aos *support vectors*. A ferramenta utilizada para maximização em cada máquina é a SVM$^{\text{Light}}$

Trabalho	Ferramenta	Dispositivo	Portável	Desempenho	Acurácia
(LU; ROY-CHOWDHURY; VANDENBERGHE, 2008)	SVM^{Light}	Máquinas conectadas em uma rede de computadores	Sim (máquinas heterogêneas)	Melhora (porém não precisa)	Melhora (aumenta 1.16% no pior caso)
(CATANZARO; SUNDARAM; KEUTZER, 2008)	LIBSVM	GPU	Não (Apenas GPUs da NVIDIA)	Melhora (*speedup* de 8.8× no pior caso)	Inaltera (mesma precisão)
(ATHANASOPOULOS et al., 2011)	LIBSVM	GPU	Não (Apenas GPUs da NVIDIA)	Melhora (porém não precisa)	Inaltera (mesma precisão)
Este trabalho	LIBSVM	GPU	Sim (qualquer GPU compatível com OpenCL)	Melhora (*speedup* de 1,45× no pior caso, após vencido o tempo de comunicação)	Piora (decresce 61,34% no pior caso; ver seção 4.2)

Tabela 2.9 – Trabalhos relacionados que paralelizam a *Support Vector Machine* com o intuito de melhorar seu desempenho.

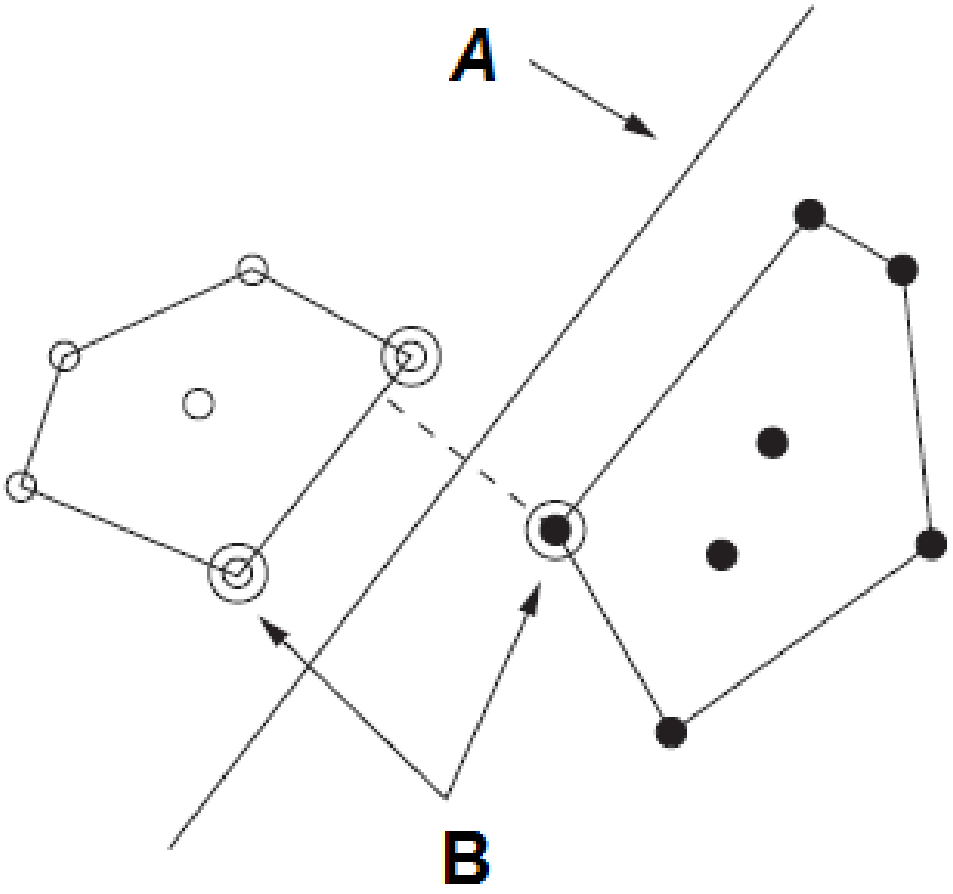

Figura 2.4 – Um hiperplano maximizado (A) contido entre os limites de dois *support vectors* (B), de classes opostas. Adaptado de (WITTEN; FRANK; HALL, 2011).

(JOACHIMS, 1998).

Em (ATHANASOPOULOS et al., 2011), a função kernel (SVM) é paralelizada através de uma implementação híbrida, onde o produto escalar entre registros do *dataset* é calculado na GPU e seu resultado é retornado para a Radial Basis Function (RBF), calculada em *threads* de CPU. Segundo o autor, tal abordagem foi adotada de forma manter a precisão do modelo preditivo gerado constante entre a LIBSVM sequencial e a implementação proposta.

Para (CATANZARO; SUNDARAM; KEUTZER, 2008), a paralelização é compatível com todos os tipos de kernel (SVM) disponíveis na LIBSVM, pelo fato que a modificação é realizada na etapa de maximização da distância do hiperplano em relação aos *support vectors*, posterior a projeção dos dados. A implementação, porém, sofre com problemas de precisão, pela LIBSVM utilizar precisão de ponto duplo e a GPU utilizada no trabalho possuir apenas precisão de ponto simples.

Tanto para (CATANZARO; SUNDARAM; KEUTZER, 2008) quanto para (ATHANA-

SOPOULOS et al., 2011), o *framework* utilizado foi o NVIDIA CUDA (NVIDIA, 2014b), específico para GPUs desta fabricante. Com isso, as implementações propostas não podem ser executadas em outros *hardware*s gráficos, sobretudo da AMD. Essa limitação é superada quando o OpenCL é utilizado.

2.5 Resumo

Neste capítulo foram explicados a configuração das placas gráficas modernas, capazes de desempenhar tarefas diversas que não apenas a renderização de gráficos, na seção 2.1; o *framework* OpenCL, capaz de executar em diversas arquiteturas de dispositivos, incluindo as GPUs, e escolhido para a paralelização da Support Vector Machine na seção 2.2; A *Support Vector Machine*, um método de aprendizado de máquina que realiza tarefas de classificação e tem se popularizado ultimamente, na seção 2.3; e finalmente, trabalhos que vêm otimizando a *Support Vector Machine* através da paralelização na GPU, sobretudo através do *framework* CUDA.

3 METODOLOGIA

Através do emprego de uma técnica de análise do funcionamento do código fonte, também conhecida como *profiling*, pretende-se encontrar trechos do código fonte que implementa a SVM e que apresentem possibilidade de paralelização. Como demonstrado em (CATANZARO; SUNDARAM; KEUTZER, 2008), (LU; ROYCHOWDHURY; VANDENBERGHE, 2008) e (ATHANASOPOULOS et al., 2011), a SVM apresenta diversos trechos paralelizáveis, o que torna a busca por eles viável. Os trechos que forem mais frequentemente utilizados pela aplicação serão preferidos aos outros, pois é preciso sobrepujar o tempo de transferência de dados da memória principal para a memória da GPU.

Uma vez encontrados os trechos mais atrativos em termos de tempo de processamento, uma primeira implementação de código paralelo é iniciada. Não se pretende alcançar um desempenho superior a SVM sequencial neste primeiro momento, mas somente explorar a viabilidade do código fonte gerado. Um segundo *profiling* identifica os gargalos de processamento, e a partir disso o código fonte deve ser refinado de forma a adequar-se melhor ao *hardware* da GPU, aumentando seu desempenho. Este processo é repetido até se atingir um desempenho satisfatório, superior ao desempenho da SVM sequencial.

3.1 Escolha e configuração de ferramentas

Uma das primeiras atividades desempenhadas diz respeito à escolha e configuração das ferramentas utilizadas. Como um dos objetivos do trabalho é disponibilizar a SVM paralelizada ao maior número de arquiteturas de GPU possível, o *framework* OpenCL mostrou-se mais adequado para escrita e execução do código fonte paralelo. Dois computadores com configurações heterogêneas foram utilizados para testar a portabilidade do código fonte gerado. A configuração dos dois computadores é citada na Tabela 3.1, onde a configuração de cada computador é descrita em uma coluna, sendo a primeira a lista de características.

Como neste trabalho são utilizadas GPUs das duas principais fabricantes, AMD (AMD, 2013) e NVIDIA (NVIDIA, 2014b), são abordadas suas versões de OpenCL; todavia, existem implementações para outros tipos de arquitetura. As configurações específicas das GPUs são descritas na Tabela 3.2, com cada GPU descrita em uma coluna e a primeira coluna contendo as características. O código fonte gerado pode apresentar alterações de desempenho de uma GPU para outra, mesmo excetuado-se todas as outras variáveis (como as especificações de

Característica	Computador A	Computador B
Sistema operacional	Windows 7 Ultimate	Windows 7 Ultimate
Conjunto de instruções	64 bits	64 bits
Processador	AMD FX 4100	AMD Phenom X4 9950
Frequência do processador	3700 MHz	2600 MHz
Cores do processador	4	4
Placa-mãe	Asus M5A88-M	Foxconn Destroyer
Memória	2x 4GB (DDR3 1333 MHz)	2x 4GB (DDR2 667 MHz)
Placa de vídeo	AMD Radeon HD 6850	NVIDIA GeForce GTX 285
Armazenamento secundário	HD Seagate ST2000DL003	HD Samsung HD103SJ
Capacidade	2 TB	1 TB

Tabela 3.1 – As duas configurações de computadores utilizados no desenvolvimento do código fonte paralelo.

Característica	AMD Radeon HD 6850	NVIDIA GeForce GTX 285
Memória	1 GB	1 GB
Tipo de memória	GDDR5	GDDR3
Banda de memória máxima	128 Gb/s	159 Gb/s
Largura de bus	256 bits	512 bits
Clock efetivo de memória	1.200 MHz	2.484 MHz
Versão do OpenCL	1.2	1.1
Tamanho máximo do work group	256	512
Stream processors	960	240

Tabela 3.2 – Características das GPUs utilizadas para execução do código fonte paralelo.

hardware dos computadores utilizados neste trabalho). Contudo, o objetivo principal é que a SVM paralelizada apresente um desempenho superior a sua contraparte sequencial, quando executadas na mesma máquina.

Para desenvolvimento e *profiling* de código fonte, o ambiente Microsoft Visual Studio 2012 (MICROSOFT, 2014a) foi escolhido. Além de conter um compilador específico para o sistema operacional Windows 7, ele é um dos ambientes de desenvolvimento compatíveis com a implementação do OpenCL da AMD e exigido pela documentação da NVIDIA (NVIDIA, 2014b).

Por fim, o código fonte da SVM foi adquirido do site da LIBSVM (CHANG; LIN, 2011). A LIBSVM é escrita em diversas linguagens, mas sua implementação original é em C/C++, o que é conveniente para este trabalho.

3.2 Estruturação do código fonte

Para facilitar o processo de otimização pela qual a LIBSVM passa e manter um registro das diferentes versões de código fonte (paralelo e sequencial) disponíveis, optou-se por criar um projeto para cada componente da LIBSVM, e agrupar os projetos em uma solução. Segundo a documentação do Visual Studio (MICROSOFT, 2014a), um projeto é um conjunto de arquivos de código fonte que constituem uma aplicação, e uma solução é um conjunto de projetos.

A LIBSVM é distribuída apenas na forma de arquivos de código fonte, sendo necessário criar os arquivos de projeto e solução do Visual Studio. É possível dividi-la em três aplicações: treino, predição e visualização. As aplicações de treino e predição mimetizam os processos realizados pela SVM teórica, enquanto a aplicação de visualização fornece uma saída gráfica para o processo classificatório da SVM; portanto, tal aplicação foi descartada da solução do Visual Studio.

A solução sobre a qual a paralelização da LIBSVM foi conduzida é composta por 4 projetos: SVMTrain (treino) e SVMPredict (predição), contendo o código fonte original, e PSVMTrain (treino paralelo) e PSVMPredict (predição paralela), constituídas pelo código fonte paralelizado.

3.3 Profiling do código fonte sequencial

O *profiling* é um método dinâmico de análise de código fonte executado por um programa, o *profiler*. Através da incorporação do código fonte a ser analisado, o *profiler* gera um perfil (*profile*) ao fim de sua execução, contendo informações como pilha de chamada e tempo de CPU das funções. Para a análise da SVM sequencial, a ferramenta de *profiling* do Visual Studio foi utilizada para gerar um *profile* das duas aplicações passíveis de paralelização, treino e predição. O *profiling* foi executado nos dois computadores, utilizando cinco *datasets* de classificação providos pela UCI *Machine Learning Repository* (BACHE; LICHMAN, 2013). A primeira coluna da Tabela 3.3 descreve as características, e as colunas subsequentes contêm as informações dos *datasets*. O resultado dos *profilings* é descrito na Tabela 3.4, com cada *dataset* sendo descrito nas colunas e a primeira coluna listando as características analisadas.

Na Tabela 3.4 é perceptível que o treino tem, em média, um tempo de execução maior que a predição. Além disso, outra informação fica implícita: o uso de CPU da aplicação de treino é mais dependente do tamanho do *dataset* do que a aplicação de predição. Isso fica

Característica	a1a	a6a	a7a	a8a	a9a
Classes	2	2	2	2	2
Atributos	123	123	123	123	123
Número de dados de treino	1.605	11.220	16.100	22.696	32.561
Número de dados de predição	30.956	21.341	16.461	9.865	16.281

Tabela 3.3 – Datasets utilizados no profiling da SVM sequencial.

Dataset	Tipo	Computador	Tempo total de execução	Função mais utilizada	Uso da função mais utilizada
a1a	Treino	A	737,23 ms	Kernel::dot	47,28%
a6a	Treino	A	22.716,04 ms	Kernel::dot	56,06%
a7a	Treino	A	44.249,79 ms	Kernel::dot	57,95%
a8a	Treino	A	89.285,76 ms	Kernel::dot	59,82%
a9a	Treino	A	188.372,10 ms	Kernel::dot	61,09%
a1a	Predição	A	7.615,39 ms	Kernel::k_function	76,46%
a6a	Predição	A	5.266,73 ms	Kernel::k_function	76,35%
a7a	Predição	A	3.723,11 ms	Kernel::k_function	76,80%
a8a	Predição	A	2.667,78 ms	Kernel::k_function	77,05%
a9a	Predição	A	3.719,35 ms	Kernel::k_function	76,69%
a1a	Treino	B	659,96 ms	Kernel::dot	28%
a6a	Treino	B	28.263,17 ms	Kernel::dot	59,76%
a7a	Treino	B	58.219,46 ms	Kernel::dot	61,33%
a8a	Treino	B	111.242,57 ms	Kernel::dot	66,18%
a9a	Treino	B	220.315,53 ms	Kernel::dot	65,67%
a1a	Predição	B	11.007,74 ms	Kernel::k_function	79,13%
a6a	Predição	B	40.043,34 ms	Kernel::k_function	85,88%
a7a	Predição	B	45.548,04 ms	Kernel::k_function	87,06%
a8a	Predição	B	32.670,73 ms	Kernel::k_function	86,51%
a9a	Predição	B	70.162,02 ms	Kernel::k_function	86,45%

Tabela 3.4 – Resultado do profiling das duas aplicações da SVM, com os cinco datasets, pelos dois computadores.

evidente colocando-se as informações da tabela 3.4 em dois gráficos, um para cada computador - Figura 3.1 para o Computador A e no Figura 3.2 para o Computador B.

Pelo fato da aplicação de treino ficar mais tempo executando que a predição, essa é escolhida para o processo de paralelização. A aplicação de predição, então, não será paralelizada; sua versão sequencial será posteriormente utilizada para verificar a coesão dos resultados do treinamento paralelizado.

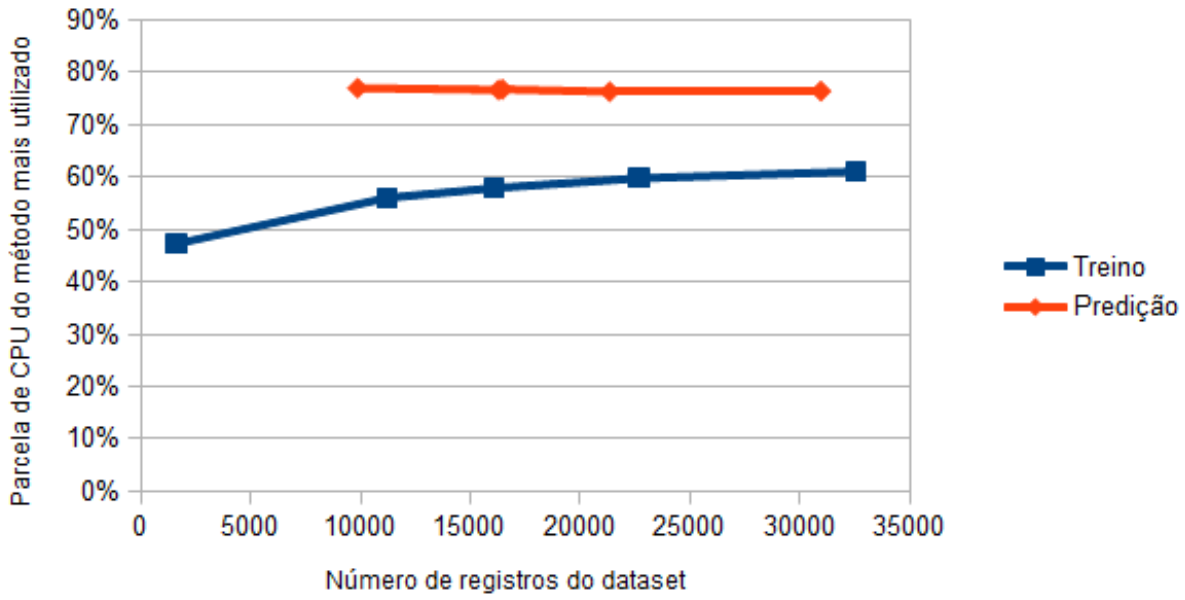

Figura 3.1 – Relação entre o tempo de CPU da função mais utilizada *versus* o tamanho do *dataset*, no Computador A.

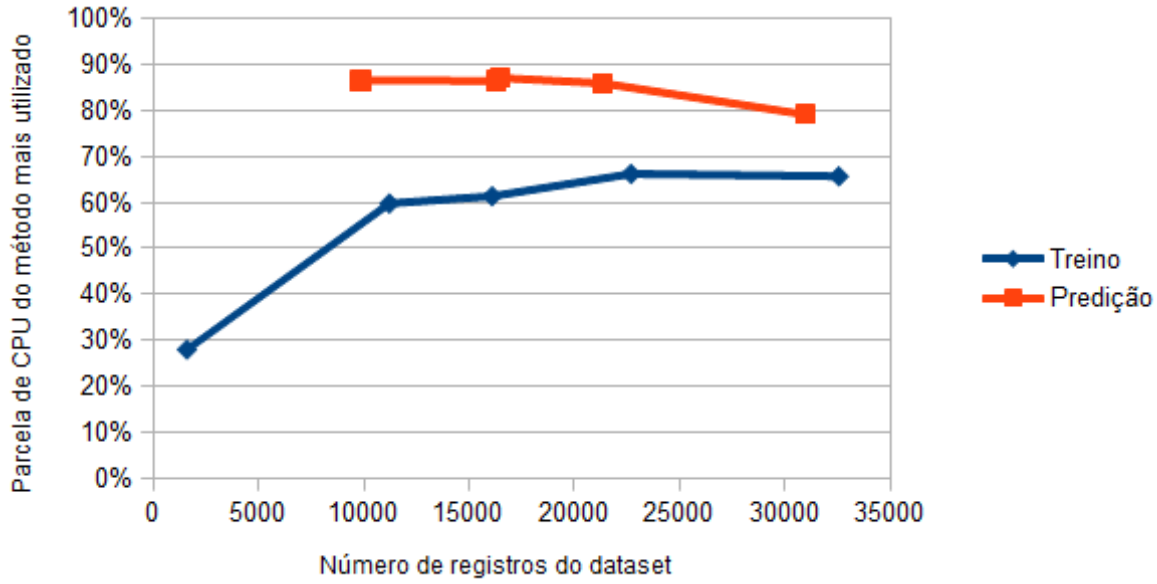

Figura 3.2 – Relação entre o tempo de CPU da função mais utilizada *versus* o tamanho do *dataset*, no Computador B.

3.4 LIBSVM

Como a paralelização da SVM será executada sobre a biblioteca que a implementa, a LIBSVM, é interessante compreender o funcionamento de seu código fonte. O objetivo é localizar uma função que itere Kernel::dot. Pela análise do código fonte, foi constatado que SVC_Q::get_Q itera Kernel::kernel_rbf, que por sua vez faz uma chamada para a função Ker-

nel::dot. SVC_Q::get_Q, então, não será paralelizada; é nessa função que o OpenCL será invocado. Kernel::kernel_rbf e Kernel::dot, por sua vez, terão seus códigos fonte transferidos para a GPU.

A função mais utilizada na etapa de treinamento, Kernel::dot, calcula o produto escalar entre dois vetores. Os vetores sobre os quais ele trabalha são os dados do *dataset*, representados no código fonte por um vetor de estruturas svm_node. A estrutura svm_node é descrita na Figura 3.3, e a função Kernel::dot na figura 3.4.

```
1. struct svm_node {
2.     int index;
3.     double value;
4. };
```

Figura 3.3 – Estrutura svm_node, composta por uma variável que representa o índice do atributo no *dataset* (index) e outra para o valor do atributo (value).

```
1.  double Kernel::dot(const svm_node *px, const svm_node *py) {
2.      double sum = 0;
3.      while (px->index != -1 && py->index != -1) {
4.      if (px->index == py->index) {
5.          sum += px->value * py->value;
6.          ++px;
7.          ++py;
8.      } else {
9.          if (px->index > py->index)
10.             ++py;
11.         else
12.             ++px;
13.         }
14.     }
15.     return sum;
16. }
```

Figura 3.4 – Função Kernel::dot, que realiza o produto escalar entre dois vetores. Cada vetor representa um registro do *dataset*, e é composto pelos atributos (representados pela estrutura svm_node) com valor não nulo (diferente de zero).

Uma svm_node é a representação computacional do valor de um atributo de um registro do *dataset*, onde index é o índice do atributo e value seu valor. Um vetor de svm_node representa, então, um registro por inteiro. O número de svm_node por vetor recebido pela função Kernel::dot varia de acordo com o número de atributos que possuem value não nulo. Nos *da-*

tasets utilizados neste trabalho, foi constatado que os únicos valores de atributo utilizados são 0 e 1. Uma consulta a outros *datasets* providos pelo site da LIBSVM (CHANG; LIN, 2011) verificou que estes não são os únicos valores possíveis, porém a particularidade do valor binário é explorada e explicada na seção 3.5.

A função Kernel::kernel_rbf representa a Radial Basis Function (RBF na sigla em inglês), um tipo de kernel (SVM) utilizado para projetar os dados do espaço de entrada para o espaço de características. A LIBSVM nem sempre utiliza a RBF; o tipo de atividade desempenhada (classificação ou regressão) é o principal determinante na escolha da função kernel (SVM). O código fonte da Kernel::kernel_rbf pode ser visto na Figura 3.5.

```
1. double kernel_rbf(int i, int j) const {
2.     return exp(-gamma*(x_square[i] + x_square[j] - 2 * dot(x[i], x[j])));
3. }
```

Figura 3.5 – Função Kernel::kernel_rbf, que representa a Radial Basis Function. A função dot visualizada acima é a função Kernel::dot.

Da função SVC_Q::get_Q, é necessário saber apenas onde a invocação da função Kernel::kernel_rbf é realizada, e então substituir o código fonte sequencial pelo paralelo. Com essas informações, é possível redigir o código fonte que invocará o OpenCL.

3.5 Paralelização

O processo de paralelização pode ser dividido em três etapas: preparação dos dados que irão para a GPU, conversão das funções que serão paralelizadas, tornando-as compreensíveis para o kernel (OpenCL), e especificação dos parâmetros de execução do OpenCL.

3.5.1 Preparação dos dados

A paralelização da LIBSVM começa pela preparação dos dados que serão enviados para a função kernel (OpenCL) e transferência da memória principal para a memória da GPU. A maioria dos dados enviados são os mesmos utilizados como parâmetros pelas funções Kernel::kernel_rbf e Kernel::dot, com exceção do parâmetro gamma para a função Kernel::kernel_rbf e dos vetores de svm_node para a Kernel::dot.

Como os *datasets* utilizados neste trabalho apresentam apenas valores de atributo binários, é possível representá-los como bits de um tipo de variável em C++, como char. Os *datasets*

utilizados possuem 123 atributos, e cada variável char tem 8 bits (ou um byte); isto faz com que sejam necessários no máximo 16 bytes para armazenar todos os valores de atributos de um registro. Na Tabela 3.5 é possível visualizar o efeito prático da modificação da representação dos dados, resultando em uma diminuição de 98,37% no espaço em memória necessário para armazenar os *datasets*. Na Tabela 3.6 os tempos teóricos de transferência de dados para esta nova representação são expressos. As variáveis char são armazenadas concomitantemente aos vetores de svm_node e são iniciadas no mesmo trecho de código fonte que estes. O uso das variáveis char não suprime os vetores de svm_node, pois enquanto essas são utilizadas na GPU estes são usados no restante do código fonte da LIBSVM.

Dataset	a1a	a6a	a7a	a8a	a9a
Registros	1.605	11.220	16.100	22.696	32.561
Atributos	123	123	123	123	123
Double para cada valor	1.579.320 bytes	11.040.480 bytes	15.842.400 bytes	22.332.864 bytes	32.040.024 bytes
Float para cada valor	789.660 bytes	5.520.240 bytes	7.921.200 bytes	11.166.432 bytes	16.020.012 bytes
Bit para cada valor	25.680 bytes	179.520 bytes	257.600 bytes	363.136 bytes	520.976 bytes

Tabela 3.5 – Tamanho da representação dos *datasets* em memória se utilizados variáveis double, float ou bits de char. Todas as variáveis são da linguagem de programação C++, e os tamanhos foram providos por (MICROSOFT, 2014b).

Dataset	a1a	a6a	a7a	a8a	a9a
Tamanho de representação (bit de char)	25.680 bytes	179.520 bytes	257.600 bytes	363.136 bytes	520.976 bytes
Transferência Computador A	200,6 ns	1.403 ns	2.013 ns	2.837 ns	4.070 ns
Transferência Computador B	161,5 ns	1.129 ns	1.620 ns	2.284 ns	3.277 ns

Tabela 3.6 – Tempos de transferência teóricos dos *datasets* para a GPU, utilizando um bit para cada valor de atributo dos registros dos *datasets*. Os tempos são expressos em nanosegundos (ns).

Essa nova abordagem de armazenamento de valores de atributo permite que a função Kernel::dot seja substituída por um cálculo AND entre as variáveis char, seguido de um somatório dos bits com valor 1. O OpenCL 1.2 possui uma função denominada popcount que faz o somatório dos bits de valor 1; para o Computador 2, que possui uma versão anterior do OpenCL, a função popcount foi implementada.

Para a função Kernel::kernel_rbf, a única modificação foi o *downcast* de double para float da variável gamma. Isto ocorre pelo fato de nenhuma das GPUs utilizadas neste trabalho possuírem suporte à extensão cl_khr_fp64 do OpenCL, que permite cálculos com variáveis double. Posteriormente observou-se que o *downcast* causou uma perda de precisão no modelo preditivo gerado, explicada na seção 4.2.

3.5.2 Conversão de funções

A transformação da função Kernel::kernel_rbf e Kernel::dot consiste em expressar o código fonte original de uma forma que possa ser compreendida pela GPU. A única função (com exceção de Kernel::dot) que é utilizada dentro do corpo de Kernel::kernel_rbf é a função matemática exp. Esta função retorna o resultado de um número elevado na base e (base natural). A implementação da função exp do kernel (OpenCL) é diferente da implementação da biblioteca math.h, utilizada pela LIBSVM, o que pode causar uma alteração no valor retornado. O OpenCL provê uma variante denominada native_exp, geralmente mais precisa que a função exp padronizada, porém com uma implementação que varia de acordo com a GPU utilizada. Constatou-se posteriormente que a utilização da native_exp foi menos responsável pelos valores retornados pela função paralela do que o *downcast* do parâmetro gamma, apesar de ainda ter uma participação importante.

A função kernel (OpenCL) é demonstrada na figura 3.6. Essa função é executada na GPU e invocada na função SVC_Q::get_Q. Os dados são transferidos para a GPU tão logo estejam disponíveis, em trechos de código fonte da LIBSVM anteriores a função SVC_Q::get_Q, para evitar uma possível espera na transferência dos dados para a GPU. Os dados são desalocados apenas ao final da execução da aplicação de treinamento.

3.5.3 Parâmetros de execução

Uma vez especificados os dados e convertidas as funções, é preciso especificar os parâmetros que possam extrair o máximo de paralelismo da GPU. O OpenCL define um *work group* como um conjunto de *work items*, sendo cada *work item* uma *thread* da GPU. Os *work items* executam concorrentemente dentro de um *work group*, e estes executam paralelamente entre si.

Pela definição da função kernel (OpenCL), cada *work item* executará uma iteração da Kernel::kernel_rbf realizada na função SVC_Q::get_Q. O OpenCL exige que o número de *work items* por *work group* seja um múltiplo do número total de *work items* que serão executa-

```
__kernel void run_rbf_getQ(
                            __global Qfloat *out,                   //vetor
                            __global int *choose,                   //escalar
                            __global int *start,                    //escalar
                            __global float *gamma,                  //escalar
                            __global schar *y,                      //vetor
                            __global float *x_square,               //vetor
                            __global int *str_len,                  //escalar
                            __global unsigned char *node_full) {    //vetor

    int
        n,
        sum = 0,
        thread_index = get_global_id(0);

    __global unsigned char
        *node_full_pivot = node_full + (*choose * *str_len),
        *node_full_change = node_full + ((thread_index + *start) * *str_len);

    __global schar
        *y_pivot = y + *choose,
        *y_change = y + (thread_index + *start);

    __global float
        *x_square_pivot = x_square + *choose,
        *x_square_change = x_square + (thread_index + *start);

    for(n = 0; n < (*str_len); n++) {
        sum += popcount(
            (*node_full_pivot) & (*node_full_change)
        );

            node_full_pivot++;
            node_full_change++;
    }

    out[thread_index] =
        (*y_pivot) * (*y_change) * (
            native_exp(
                (-(*gamma)) * (
                    (*x_square_pivot) + (*x_square_change) - 2 * sum
                )
            )
        );
}
```

Figura 3.6 – Função kernel (OpenCL) que implementa as funções Kernel::kernel_rbf e Kernel::dot.

dos na GPU. Como cada GPU possui um limite do número de *work items* por *work group* e existe uma possibilidade de que o tamanho do *dataset* seja um número primo, optou-se por mandar para a GPU o mesmo número de *work items* e *work groups*. Na prática, são enviados $\sqrt{tamanho\ do\ dataset}$ *work groups* com $\sqrt{tamanho\ do\ dataset}$ *threads* cada, totalizando $(\sqrt{tamanho\ do\ dataset})^2$ *threads*. É importante salientar que o cálculo é feito com números inteiros, portanto o número de *threads* executadas na GPU não é o mesmo que o tamanho do *dataset*. Para o *dataset* a1a, que possui 1605 registros, por exemplo, são enviados 40 *work groups*, cada um com 40 *threads*, totalizando 1600 *threads* na GPU. As últimas 5 iterações de Kernel::kernel_rbf foram executadas na CPU, com o código fonte original.

4 RESULTADOS OBTIDOS

O código fonte paralelizado foi armazenado em um repositório Git, e pode ser conferido em https://hcagnini@bitbucket.org/hcagnini/parallel_libsvm_train.git. As modificações realizadas na LIBSVM foram capazes de aumentar seu desempenho, ao custo de uma piora na precisão dos modelos preditivos gerados.

4.1 Desempenho

O aumento de desempenho é maior a medida que o tamanho do *dataset* cresce. Os tempos de execução precisos podem ser visualizados nas tabelas 4.1 para o Computador A e 4.2 para o Computador B, bem como o *speedup* (vezes que o código fonte paralelo é mais rápido que o código fonte sequencial). Como o *dataset* a1a foi o único a obter uma piora no seu tempo, o valor de seu *speedup* é expresso por um número negativo - significando então que houve um ganho no tempo total.

Dataset	Tempo de execução sequencial	Tempo de execução paralelo	*speedup*
a1a	737,23 ms	1.707,19 ms	-2,32×
a6a	22.716,04 ms	15.643,04 ms	1,45×
a7a	44.249,79 ms	22.119,55 ms	2×
a8a	89.285,76 ms	37.146,44 ms	2,4×
a9a	188.372,10 ms	63.669,72 ms	2,96×

Tabela 4.1 – Tempos de execução das versões sequencial e paralela para o Computador A, mostrando também o *speedup* no tempo de execução. O *dataset* a1a teve uma piora no desempenho com a versão paralela, sendo seu *speedup* expresso então por um número negativo.

Dataset	Tempo de execução sequencial	Tempo de execução paralelo	*speedup*
a1a	659,86 ms	688,46 ms	-1,04x
a6a	28.263,17 ms	18.537,82 ms	1,52×
a7a	58.219,46 ms	37.546,28 ms	1,55×
a8a	111.242,57 ms	46.130,17 ms	2,41×
a9a	220.315,53 ms	85.083,07 ms	2,59×

Tabela 4.2 – Tempos de execução das versões sequencial e paralela para o Computador B, mostrando também a taxa de redução do tempo de execução. O *dataset* a1a teve uma piora no desempenho com a versão paralela, sendo seu *speedup* expresso então por um número negativo.

A piora de desempenho da aplicação de treino quando utilizando o *dataset* a1a pode

ser explicada pelo fato de o *dataset* não ser grande o suficiente para compensar o tempo de transferência de dados e execução da função kernel::kernel_rbf na GPU. A diferença do ganho de tempo - 76,01% para o Computador A e 4,33% para o Computador B - é condizente, ainda, com as informações descritas na Tabela 3.2, onde a GPU do Computador B possui banda de memória maior que a GPU do Computador A.

Para os outros *datasets*, o aumento de desempenho é crescente a medida que o tamanho do *dataset* aumenta. Apesar de parecer paradoxal, a explicação é simples, e melhor entendida quando projetamos os dados das tabelas 4.1 e 4.2 para os gráficos da Figura 4.2. O tempo de execução da versão sequencial aumenta muito mais rápido que sua contraparte paralela, fazendo com que a área entre as duas funções aumente a medida que o tamanho do *dataset* também aumenta. É possível que após um certo tamanho de *dataset* o ganho de desempenho estabilize-se ou até mesmo diminua, porém isto não pode ser afirmado para os cinco *datasets* utilizados neste trabalho.

Mesmo os resultados sendo positivos, surge a dúvida se o aumento de desempenho deve-se mais à modificação da representação dos *datasets* (descrito na seção 3.5.1) do que à execução paralela do código fonte. Para responder isso, o código fonte paralelo foi modificado para execução na CPU, sequencialmente. A função exp do OpenCL foi substituída pela exp da biblioteca math.h, e a popcount pela __popcnt provida pelo código fonte do Visual Studio. As funções e suas substitutas sequenciais possuem o mesmo funcionamento, não alterando os resultados do processamento. A modificação é vista na Figura 4.1.

Apesar das modificações serem bem sucedidas e o *dataset* a1a executar em tempo hábil, para o restante dos *datasets* a aplicação executou indefinidamente. A nova representação dos dados, então, é mais adequada ao processamento paralelo, onde a aplicação tem um tempo de execução finito.

4.2 Precisão

A perda de precisão é constante dentre todos os *datasets*, com exceção do a8a para o Computador B. Dois fatores foram responsáveis pela perda: a precisão da função native_exp e o *downcast* de double para float do parâmetro gamma da função Kernel::kernel_rbf. As taxas de precisão podem ser observadas na Tabela 4.3.

Segundo a documentação do OpenCL (KHRONOS, 2012), a função native_exp é variável de acordo com a GPU, podendo ser mais ou menos precisa dependendo da implementação

```
for(j = start; j < len; j++) {
    float sum = 0;
    for(int z = 0; z < this->str_len; z++) {
        sum += __popcnt(this->node_full[i][z] & this->node_full[j][z]);
    }

    data[j] = (Qfloat)(
        y[i] * y[j] * exp(
            (-(float)gamma) * (
                x_square[i] + x_square[j] - 2 * sum
            )
        )
    );
}
```

Figura 4.1 – Função executada sequencialmente na CPU, utilizando a mesma representação de dados que a versão paralela.

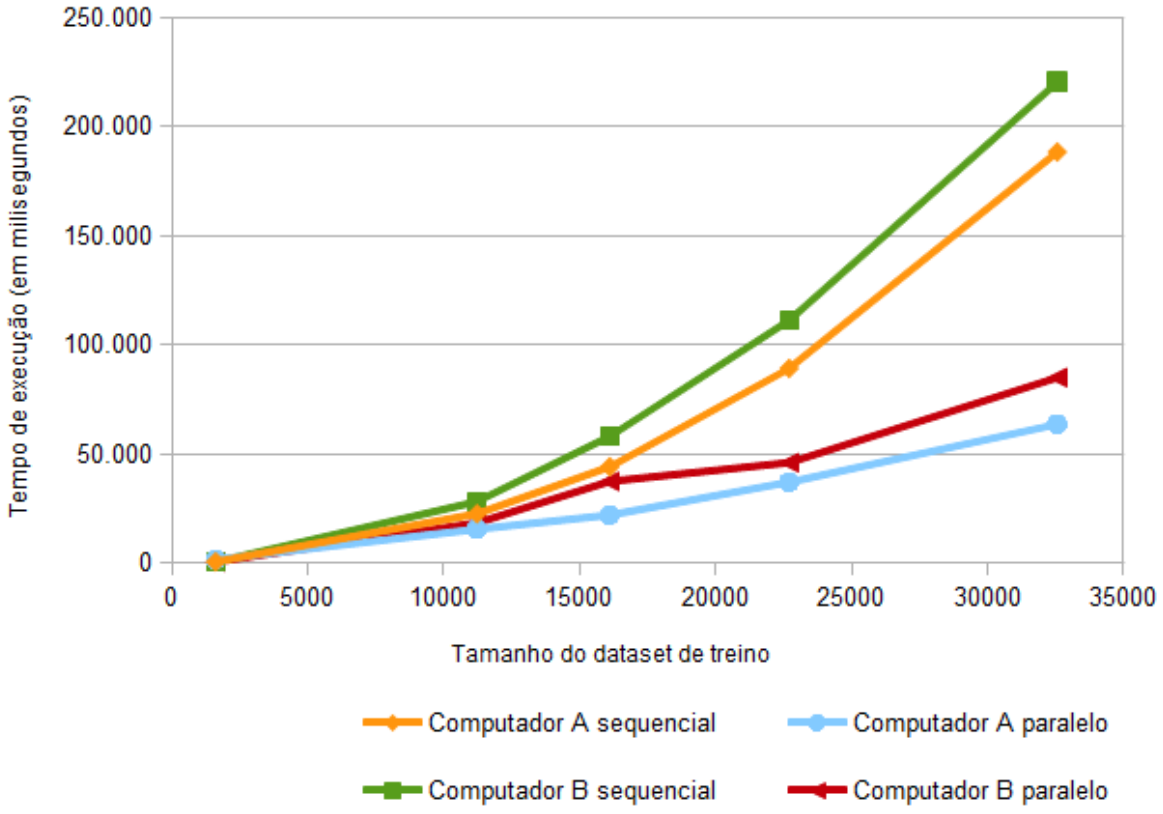

Figura 4.2 – Comparação entre o tempo de execução de cada computador, com as versões sequencial e paralela, versus o tamanho do dataset utilizado no treinamento.

feita pelo fabricante; a função exp, por sua vez, tem precisão padronizada pelo OpenCL para todas as GPUs. Durante o desenvolvimento do código fonte, a utilização da função exp tanto

para o Computador A quanto para o Computador B resultou em uma precisão em torno de 23% do modelo preditivo para o *dataset* a8a. Quando utilizada a função native_exp, a precisão do modelo do Computador A subiu para 76,33%, enquanto a do Computador B permaneceu em 23,67%. Fica implícito que o fabricante da GPU do Computador B fez a mesma implementação tanto para a função native_exp quanto para a exp.

Porém, pode parecer inconsistente responsabilizar o fabricante da GPU do Computador B pela perda quando os modelos gerados para os outros *datasets* obtiveram exatamente a mesma precisão que os modelos do Computador A. O contraponto é que o desvio ocorre na função kernel (SVM), fazendo com que os dados do espaço de entrada sejam projetados para posições incorretas no espaço de características. Além do *dataset* a8a ser mais sensível a desvios de valores gerados pela função paralela, comprovado pela diferença de precisão entre os modelos do Computador A e B substituindo apenas a função exp pela native_exp, deve-se também atentar para o fato de que o *dataset* a8a possui o menor número de dados de teste, fazendo com que qualquer dado incorretamente classificado tenha maior impacto na taxa de precisão do que os outros *datasets*.

Resta, ainda, justificar a perda de precisão que todos os *datasets* sofreram, à exceção de a1a. Isto explica-se pelo *downcast* do parâmetro gamma da função kernel::kernel_rbf de double para float: durante sessões de depuração do código fonte, constatou-se que alguns dados sofreram um desvio na oitava casa decimal. Isto é condizente com as informações apresentadas em (MICROSOFT, 2014a): a variável C++ float tem precisão até a sétima casa decimal, enquanto double possui até a décima quinta.

Dataset	Computador A sequencial	Computador A paralelo	Computador B sequencial	Computador B paralelo
a1a	83,59%	83,59%	83,59%	83,59%
a6a	84,17%	75,87%	84,17%	75,87%
a7a	84,58%	76,17%	84,58%	76,17%
a8a	85,01%	76,33%	85,01%	23,67%
a9a	84,82%	76,38%	84,82%	76,38%

Tabela 4.3 – Precisão dos modelos preditivos gerados pelos Computadores A e B, para as versões sequencial e paralela.

5 CONSIDERAÇÕES FINAIS

Neste trabalho a *Support Vector Machine* foi paralelizada, através da execução na GPU das funções mais utilizadas no código fonte da aplicação de treino da biblioteca que a implementa, a LIBSVM. O *framework* utilizado para executar o código fonte na GPU é o OpenCL. Foram testados cinco *datasets*, tanto para a versão paralelizada quanto para a versão sequencial do método de classificação. Os *datasets* foram utilizados na aplicação de treino para gerar um modelo preditivo, que é então utilizado na etapa de predição para classificar dados sobre os quais não se sabe o valor de seu atributo classe.

A versão paralelizada obteve um desempenho superior a versão sequencial, chegando a dois terços de redução para o maior *dataset*, o a9a. A implementação é ainda portável, sendo executada nas duas arquiteturas de GPU mais comuns do mercado, NVIDIA e AMD. Contudo, os modelos preditivos gerados pela versão paralela obtiveram uma piora de aproximadamente 10% na precisão da classificação em relação a versão sequencial, para todos os *datasets*. Isto se dá principalmente pelo *downcast* de variáveis do tipo double para float, imposto ao código fonte pelo fato de nenhuma das GPUs testadas possuir suporte ao tipo double.

Apesar da perda de precisão verificada na versão modificada da LIBSVM, é possível manter o desempenho adquirido com a paralelização e ainda aumentar a precisão através do emprego de um conjunto de outras técnicas de otimização, ou aprofundamento das técnicas apresentadas.

5.1 Trabalhos futuros

Durante a escrita do código fonte paralelo, foi possível constatar que a LIBSVM pode se beneficiar de uma abordagem mais agressiva de otimização, paralelizando não apenas as funções mais utilizadas como também modificando a forma como os dados são processados em todo processo classificatório da SVM. Além disso, também verificou-se que um aprofundamento das técnicas apresentadas pode melhorar a precisão mantendo ao mesmo tempo que o desempenho alcançado pela paralelização.

5.1.1 Novos parâmetros de execução

A criação de *work items* e *work groups* apresentada segue uma abordagem generalista, tentando manter o número de *work items* por *work group* igual ao número de *work groups*. Tal abordagem não leva em consideração o tempo de execução de cada *thread* da GPU. Um tempo de execução mais longo das *threads* implica em menos *work items* por *work group*, e vice-versa. Com um *profiling* direcionado ao tempo de execução das *threads* de GPU, é possível aferir este tempo e configurar os parâmetros de execução de forma a extrair um desempenho melhor.

5.1.2 Implementação da função exponencial

Uma forma de garantir uma precisão constante para a função exp executada na GPU é sobrescrevê-la por um código fonte redigido pelo usuário. Essa abordagem é semelhante ao que foi feito com a função popcount, com a diferença que o propósito era suprir a carência desta no OpenCL 1.1 da GPU do Computador B. Sobrescrevendo a função exp, é possível manipular o processo computacional e determinar uma precisão que adeque-se aos valores necessários à produção de um modelo preditivo mais preciso.

5.1.3 Utilização de GPUs com suporte a double

Nenhuma das GPUs utilizadas neste trabalho possui suporte a variáveis double, e o fato de que elas ainda são opcionais na especificação da última versão do OpenCL, a 2.0 (KHRONOS, 2014) sugere que o suporte não será universal nas placas de vídeo mais recentes no mercado. Para modelos preditivos mais sensíveis aos dados de entrada, sugere-se então a utilização de uma GPU com o suporte a variável double, eliminando o problema do *downcast* do parâmetro gamma da função Kernel::kernel_rbf. Porém, como analisado em (AMD, 2013), a utilização de double no kernel (OpenCL) acarreta em uma depreciação no desempenho da aplicação em que elas estão são utilizadas. Seria preciso então analisar a proporção de perda de processamento *versus* ganho de precisão e julgar se a relação justifica a utilização de uma GPU com suporte a double.

5.1.4 Pré-processamento do dataset

Como explicado na seção 3.5, os *datasets* utilizados neste trabalho apresentam apenas valores de atributo binários, possibilitando uma representação de dados diferente da original

no código fonte da LIBSVM. Apesar de isso constituir uma limitante imediata a *dataset*s que não apresentem estes valores, existem diversas aplicações que convertem os dados de um *dataset* para o formato aceito pela LIBSVM, como o Weka (WITTEN; FRANK; HALL, 2011) e uma ferramenta provida pela própria documentação da LIBSVM (CHANG; LIN, 2011). Modificando-se estas aplicações de forma que os dados do *dataset* no formato da LIBSVM sejam sempre expressos em formato binário, torna-se possível utilizar a versão paralelizada da LIBSVM.

Da mesma forma, para preservação da estrutura do *dataset*, é possível também substituir a abordagem binária por uma de multiplicação de matrizes, como descrita em (ATHANASOPOULOS et al., 2011). Independente da proposta, deve ser analisado o custo de implementação de modificações e o ganho previsto no desempenho para ambas implementações.

GLOSSÁRIO

atributo Uma informação específica sobre um registro de um conjunto de dados. Um conjunto de dados pode conter vários atributos. Um atributo preditivo é uma informação auxiliar na descoberta do valor do atributo classe, sempre presente em um conjunto de dados. 18, 19, 30

atributo classe É a informação mais importante de um conjunto de dados, e frequentemente a razão pela qual se agrupam dados em conjuntos: para saber o valor do atributo classe. Os demais atributos de um conjunto de dados podem ou não estar relacionados com o valor do atributo classe. 18, 19, 20, 40

clock A menor porção de tempo de um processador. 11

conjunto de dados Formado por diversos dados agrupados em um arquivo. Cada dado também pode ser denominado por registro. Um conjunto pode ser composto por dados das mais diversas naturezas, desde valores coletados por um sensor até uma pesquisa de opinião, por exemplo. 15

CPU Sigla em inglês para Central Processing Unit. 10, 12, 23, 27, 33, 37

dado Relativo a qualquer tipo de contexto – a opinião de uma pessoa sobre um determinado assunto, valores de um sensor, etc. Um dado é composto por vários atributos e por um atributo classe, que pode ou não ter seu valor definido. 13, 18, 19, 20

dataset Termo em inglês para conjunto de dados. 13, 15, 18, 19, 20, 23, 27, 30, 31, 33, 36, 37, 39, 40, 41, 42

espaço de características Um espaço com dimensionalidade maior que o espaço de entrada, onde os dados podem ser linearmente separados por um hiperplano. 19

espaço de entrada O espaço dimensional onde os dados se encontram originalmente. Pode ou não permitir a separação linear através de um hiperplano. Caso não permita, deve-se mapear os dados para um espaço com dimensionalidade suficiente para permitir a separação. Tal espaço é denominado espaço de características. 19

feature space Termo em inglês para espaço de características. 19

framework trata-se do conjunto de bibliotecas, padronizações e metodologia sobre a qual um trabalho será realizado. O framework OpenCL, por exemplo, oferece bibliotecas em C/C++ que permitem o acesso da GPU pelo código-fonte do programador e estabelece um padrão como tal acesso deve ser realizado. 10, 11, 12, 25

General Purpose Graphics Processor Unit Designação específica para GPUs que são capazes de desempenhar outras funções que não apenas a renderização de gráficos. O termo vem caindo em desuso pelo fato de atualmente a maioria das GPUs permitirem tarefas genéricas em seu hardware. 10, 11

GPGPU sigla para General Purpose Graphics Processor Unit. 10, 11

GPU sigla para Graphics Processor Unit. 10, 11, 12, 13, 23, 24, 25, 28, 31, 32, 33, 36, 37, 39, 40, 41

Graphics Processor Unit Hardware dedicado principalmente a renderização de gráficos, mas que também pode ser utilizado para outras tarefas computacionais. 11

kernel (SVM) Uma função com o objetivo de mapear os dados no espaço de entrada (input space) para um espaço de características (feature space), onde possam ser separados linearmente por um hiperplano. 23, 31, 39

kernel (OpenCL) Uma função que primeiramente é pensada para ser executada em um dispositivo diferente da CPU, mas que – pela flexibilidade do OpenCL – pode também ser executado por ela, caso não exista nenhum dispositivo elegível para execução. 12, 13, 31, 33, 41

profile Artefato gerado por um profiler ao fim de um processo de profiling; em outras palavras, um relatório contendo a pilha de chamada de função, alocação de tempo de CPU, memória, dentre outras informações pertinentes ao software analisado pelo profiler. 27

profiler Um programa que analisa um outro programa e, após a execução deste, mostra resultados como uso de memória, processador, funções mais chamadas pelo código fonte, etc. 27

profiling Ação desempenhada pelo profiler. Tem como objetivo produzir um perfil (profile). 25, 26, 27, 40

registro Uma entrada em um conjunto de dados. Também pode ser simplesmente chamado de dado. 18, 19, 30, 31

support vector Uma reta constituída por dados na fronteira de uma classe. 19

Support Vector Machine Um método de aprendizado de máquina, proposto em (CORTES, VAPNIK, 1995). 9, 12, 13, 14, 19, 21, 24, 40

SVM Sigla para Support Vector Machine. 9, 10, 11, 13, 19, 20, 25, 26, 27, 28, 40

REFERÊNCIAS

AMD. **AMD Accelereated Parallel Processing - OpenCL Programming Guide**. Disponível em <http://developer.amd.com/wordpress/media/2013/07/AMD_Accelerated_Parallel_Processing_OpenCL_Programming_Guide-rev-2.7.pdf>, Acesso em: 13 mai 2014.

ATHANASOPOULOS, A. et al. GPU Acceleration for Support Vector Machines. **Proceedings of 12th International Workshop on Image Analysis for Multimidia Interactive Services**, Delft, Holanda, 2011.

BACHE, K.; LICHMAN, M. **UCI Machine Learning Repository**. Disponível em <http://archive.ics.uci.edu/ml>, Acesso em: 08 mai 2014.

CATANZARO, B.; SUNDARAM, N.; KEUTZER, K. Support Vector Machine Training and Classification on Graphics Processors. **Proceedings of the 25th International Conference on Machine Learning**, Helsink, Finlândia, p.104–111, 2008.

CHANG, C.; LIN, C. LIBSVM: a library for support vector machines. **ACM Transactions on Intelligent Systems and Technology**, [S.l.], v.2, p.27, 2011.

DE SOUZA, B. F. **Seleção de Características em SVMs Aplicadas a Dados de Expressão Gênica**. 2005. Dissertação (Mestrado em Ciência da Computação) — Instituto de Ciências Matemáticas e de Computação, São Carlos, São Paulo.

ELSEVIER. **Scopus**. Disponível em <http://www.scopus.com>, Acesso em: 20 jun 2014.

FOSTER, I. **Designing and building parallel programs**: conceps and tools for parallel software enginering. Disponível em <http://www-unix.mcs.anl.gov/dbpp>, Acesso em: 12 mai 2014.

HARDING, S.; BANZHAF, W. Fast Genetic Programming on GPUs. **10th European Conference on Genetic Programming**, [S.l.], v.4445, p.90, 2007.

JOACHIMS, T. Making large-scale SVM learning practical. **Advances in Kernel Methods - Support Vector Learning**, [S.l.], p.169–184, 1998.

KARIMI, K.; DICKSON, N.; HAMZE, F. A Performance Comparison of CUDA and OpenCL. **ArXiv e-prints**, [S.l.], 2010. Disponível em <http://adsabs.harvard.edu/abs/2010arXiv1005.2581K>.

KHRONOS. **The OpenCL 1.2 Specification**. Disponível em <http://www.khronos.org/registry/cl/specs/opencl-1.2.pdf>, Acesso em: 11 mar 2014.

KHRONOS. **The OpenCL 2.0 Specification**. Disponível em <https://www.khronos.org/registry/cl/specs/opencl-2.0.pdf>, Acesso em: 17 jun 2014.

LU, Y.; ROYCHOWDHURY, V.; VANDENBERGHE, L. Distributed Parallel Support Vector Machines in Strongly Connected Networks. **IEEE Transactions on Neural Networks**, [S.l.], p.1167–1178, 2008.

MICROSOFT. **Visual Studio Documentation**. Disponível em <http://www.visualstudio.com/pt-br/get-started/overview-of-get-started-tasks-vs>, Acesso em: 31 mar 2014.

MICROSOFT. **Data Type Ranges**. Disponível em <http://msdn.microsoft.com/en-us/library/s3f49ktz.aspx>, Acesso em: 14 jul 2014.

NVIDIA. **CUDA Home Page**. Disponível em <http://www.nvidia.com/object/cuda_home_new.html>, Acesso em: 16 mar 2014.

NVIDIA. **CUDA Toolkit Documentation**. Disponível em <http://docs.nvidia.com/cuda/>, Acesso em: 31 mar 2014.

TECHSPOT. **The History of Modern Graphics Processor**. Disponível em <http://www.techspot.com/article/650-history-of-the-gpu/>, Acesso em: 16 abr 2014.

VAPNIK, V. An overview of statistical learning theory. **IEEE Transactions on Neural Networks**, [S.l.], v.10, p.988–999, 1999.

VAPNIK, V.; CORTES, C. Support Vector Networks. **Machine Learning**, [S.l.], v.20, p.273, 1995.

WITTEN, I.; FRANK, E.; HALL, M. **Data Mining - Practical Machine Learning Tools and Techniques**. 3.ed. [S.l.]: Morgan Kaufmann, 2011. 629p.