UNIVERSIDADE FEDERAL DE SERGIPE

CENTRO DE CIÊNCIAS EXATAS E TECNOLOGIA

DEPARTAMENTO DE COMPUTAÇÃO

CURSO DE BACHARELADO EM CIÊNCIA DA COMPUTAÇÃO

DESENVOLVIMENTO DE ALGORITMO DE OTIMIZAÇÃO POR ENXAME DE PARTÍCULAS PARA APRENDIZADO DE REGRAS DE CLASSIFICAÇÃO EM CUDA

THALES FRANCISCO SOUSA SAMPAIO ALVES-DOS-SANTOS

SÃO CRISTÓVÃO – SE

2017

THALES FRANCISCO SOUSA SAMPAIO ALVES-DOS-SANTOS

DESENVOLVIMENTO DE ALGORITMO DE OTIMIZAÇÃO POR ENXAME DE PARTÍCULAS PARA APRENDIZADO DE REGRAS DE CLASSIFICAÇÃO EM CUDA

Trabalho de conclusão de Curso apresentado ao Departamento de Computação da Universidade Federal de Sergipe como exigência para obtenção do grau de Bacharel em Ciência da Computação

Orientador: André Britto de Carvalho

SÃO CRISTÓVÃO – SERGIPE

2017

ALVES-DOS-SANTOS, T. F. S. S.

Desenvolvimento de Algoritmo de Otimização por Enxame de Partículas para Aprendizado de Regras de Classificação em CUDA / Thales Francisco Sousa Sampaio Alves dos Santos – São Cristóvão: UFS, 2017.

51f.;

Trabalho de Conclusão de Curso (graduação) – Universidade Federal de Sergipe, Curso de Bacharelado em Ciência da Computação, 2017.

1. Mineração de dados 2. Tecnologia da Informação – TCC 3. Ciência da Computação I. Desenvolvimento de Algoritmo de Otimização por Enxame de Partículas para Aprendizado de Regras de Classificação em CUDA

THALES FRANCISCO SOUSA SAMPAIO ALVES DOS SANTOS

DESENVOLVIMENTO DE ALGORITMO DE OTIMIZAÇÃO POR ENXAME DE PARTÍCULAS PARA APRENDIZADO DE REGRAS DE CLASSIFICAÇÃO EM CUDA

Trabalho de Conclusão de Curso submetido ao corpo docente do Departamento de Computação da Universidade Federal de Sergipe (DCOMP/UFS) como parte dos requisitos para obtenção do grau de Bacharel em Ciência da Computação.

São Cristóvão, 27 de outubro de 2017

Banca Examinadora:

André Britto de Carvalho

Universidade Federal de Sergipe

Bruno Otavio Piedade Prado

Universidade Federal de Sergipe

Janio Coutinho Canuto

Universidade Federal de Sergipe

Dedico este trabalho a minha esposa Juliana,
e meus filhos Rodrigo e Nina.

AGRADECIMENTOS

Em primeiro lugar agradeço a Ju, minha amada esposa, que sem o apoio total e irrestrito dela eu nunca chegaria ao fim desta jornada.

Meus sogros Alcione e Salma e minha mãe Marlene, que me ajudaram na correria do dia a dia, e me segurando as pontas sempre que necessário que sem esse apoio dificilmente eu terminaria esta jornada.

Ícaro Marley, Gabriel Araújo, Wagner Macedo e a galera do PUG-SE por ajudar nas dúvidas com python quando precisei gerar um script automatizado para execução dos experimentos.

Gustavo Woerner, por me “emprestar” a placa de vídeo dele, executando os experimentos em seu computador gamer top linha 2017 super hiper power plus ao invés de usar minha máquina velha de guerra com placa de vídeo limitada.

Rodrigo e Nina, meus filhos, apesar de todo trabalho sempre me lembrando como é bom ser criança e como posso clarear as ideias brincando e assistindo desenhos juntos.

Agradeço a Prof. André por aceitar a me orientar e pela ideia do trabalho para que eu chegasse até aqui.

RESUMO

A geração de dados aumenta a cada dia, espera-se que em 2020 existam 44 trilhões GB de dados criados e copiados. A Mineração de Dados (MD) é um processo automático ou semiautomático para tentar extrair informações relevantes em grandes volumes de dados. Neste processo de MD é realizado através de diversas tarefas, classificação, agrupamento e aprendizado de regras entre elas. Um problema de Aprendizado de Regras (AR) busca encontrar regras de associação dentro de um conjunto de dados, as regras de associação são representadas por meio de afirmações SE-ENTÃO. Ao definir como consequente da regra o rótulo da base de dados é possível gerar regras de classificação. Estas regras podem ser representadas na forma de vetores para avaliação automatizada da qualidade, para que sejam encontradas as melhores. Desta forma, o AR pode ser resolvido como um problema de otimização. Existem diversas soluções do AR com algoritmos de otimização, mas o desempenho deles tende a piorar em grandes volumes. A NVIDIA, em 2006, apresentou o CUDA, que é uma plataforma de computação paralela utilizando GPU, há uma busca de algoritmos paralelos para tratar do Big Data e com diversos pesquisadores propondo soluções em CUDA para criar algoritmos paralelos. Este trabalho buscou desenvolver um algoritmo de otimização para ser usado em AR desenvolvido em CUDA que se aproveitasse do conceito de múltiplos enxames de partículas e do processamento altamente paralelo da plataforma alvo de forma a aumentar a velocidade de processamento de bases de dados. Este algoritmo é uma extensão do algoritmo M-PSO. Para validar o novo algoritmo foi utilizada validação cruzada com 10 partições, cada algoritmo foi rodado 10 vezes sobre 7 bases de dados diferentes, obtendo uma execução de até 51,8% mais rápida com resultados de qualidade similar ao M-PSO.

Palavras-chave: Mineração de Dados, CUDA, Aprendizado de Regras, Otimização por Enxame de Partículas.

ABSTRACT

Data generation increases every day, it is expected that by 2020 there will be 44 trillion GB of data created and copied. Data Mining (MD) is an automatic or semi-automatic process for attempting to extract relevant information over large volumes of data. MD is performed through various tasks, classification, grouping and rule learning among them. A Rule Learning (AR) problem seeks to find association rules within a dataset, association rules are represented by IF-THEN statements. By defining the database label as a rule consequence, it is possible to generate classification rules. These rules can be represented in the form of vectors for automated evaluation of their quality so that the best ones can be found, in this way AR can be solved as an optimization problem. Several AR solutions have already been demonstrated with optimization algorithms, but their performance tends to worsen in large volumes. NVIDIA in 2006 introduced CUDA, which is a parallel computing platform using GPUs, and there is a search for parallel algorithms to handle Big Data, and several researchers have proposed their algorithms in CUDA to create parallel algorithms. This work aimed to develop an optimization algorithm to be used in AR developed in CUDA that took advantage of the concept of multiple particles swarms and the highly parallel processing of the target platform to increase the speed of database processing. This algorithm is an extension of the M-PSO algorithm. To validate the new algorithm we used cross-validation with 10 partitions, each algorithm was run 10 times over 7 different databases, obtaining an execution of up to 51.8% faster with results of similar quality to the M-PSO.

Keywords: Data Mining, CUDA, Rule Learning, Particle Swarm Optimization

LISTA DE ABREVIATURAS E SIGLAS

AM	Aprendizado de Máquina
AR	Aprendizado de regras
CPU	Unidade Central de Processamento
CUDA	*Compute Unified Device Architeture*
FN	*False Negative*
FP	*False Positive*
GPU	Unidade de Processamento Gráfica
KDD	*Knowledge-discovery in databases*
MD	Mineração de Dados
MHMO	Metaheuristicas Multiobjetivo
MOPSO	*Multiobjective Particle Swarm Optimization*
PSO	*Particle Swarm Optimization*
SM	Multiprocessadores de Streaming
TN	*True Negative*
TP	*True Positive*

SUMÁRIO

1 – INTRODUÇÃO

A cada dia que passa são gerados mais dados que podem ser utilizados para os mais variados fins. Este volume cresce exponencialmente e segundo Turner *et al.* (2014) é esperado que em 2020 existam 44 trilhões GB de dados criados e copiados. Segundo FACEBOOK (2016), no ano de 2015 haviam 3,2 bilhões de pessoas conectadas à internet, usando principalmente seus *smartphones* e *featurephones*, que estão constantemente gerando novos dados, seja pela simples conexão à internet até o uso cada vez mais intensivo de redes sociais. Além destas pessoas existem aparelhos conectados à Internet oferecendo e consumindo dados que crescem a cada ano (Turner *et al.*, 2014). Através deste grande volume de dados é possível extrair informações que ainda não são conhecidas.

Dados são fatos, valores medidos ou calculados, que podem se encaixar num contexto para se tornar informação. Uma base de dados (BD) é um conjunto de dados com alguma relação entre si. Cada dado é composto por diversos atributos e para que estes dados possam ser categorizados um dos atributos pode ser escolhido como rótulo.

O volume de dados gerado por todos os setores da sociedade é tão grande e/ou complexo que os algoritmos tradicionais de processamento de dados são inadequados para lidar com eles, a este volume de dados é dado o nome *Big Data*. E mais recentemente o termo *Big Data* tem sido usado mais para se referir ao uso de análises preditivas, análises de comportamento de usuários ou outros métodos de análise de dados avançados que extraem informação dos dados e menos ao tamanho do banco de dados em particular.

No contexto do *Big Data*, a Mineração de Dados (MD) é utilizado como solução para o processamento e obtenção de informações em grandes volumes de dados. A MD é um processo automático ou semiautomático de exploração analítica de grandes bases de dados, na tentativa de descobrir padrões relevantes que ocorrem nestes dados e que sejam importantes para dar base à assimilação de informação importante, ajudando desta forma a geração do conhecimento (SILVA, et al. 2016).

Para capturar os padrões da MD é necessário executar algumas tarefas de aprendizagem. A classificação é uma das tarefas mais comuns, ela visa identificar a qual classe um determinado registro pertence. Na classificação o modelo analisa o conjunto de registros fornecidos com seus rótulos para aprender como classificar um novo registro, é também conhecida como aprendizado supervisionado. Agrupamento é a tarefa que visa identificar e aproximar os dados similares, ela não recebe dados pré-categorizados e por isso é um aprendizado não supervisionado. A aprendizagem de regras é a tarefa de identificar relações entre atributos.

Nestas tarefas da MD é necessário fazer o treinamento, que consiste no processamento da base de dados para a categorização das informações nelas contidas de forma que se possa prever resultados futuros. Após o treinamento faz-se um teste, que se utiliza de uma parte da base de dados já previamente categorizada e não utilizada no treinamento, verificando se o algoritmo é capaz de categorizá-las corretamente usando o modelo criado no treinamento.

Um problema de Aprendizado de Regras (AR) pode ser definido como sendo dado um número de exemplos de treinamento, encontrar um grupo de regras de associação que podem ser utilizadas para predição ou classificação de novas instâncias (FÜRNKRANZ, *et al.* 2012). Estas regras de associação são representadas por meio de afirmações do tipo SE-ENTÃO. Essas regras relacionam determinada condição a um determinado resultado, no formato SE uma condição (antecedente ou premissa) ENTÃO um resultado (consequente ou conclusão) (HAN, et al. 2012). Desta forma estas regras partem do pressuposto que a presença de algum atributo num evento implica a presença de outro no mesmo evento (SILVA, *et al.* 2016). Quando o consequente é o rótulo esta regra é denominada regra de classificação. O AR pode ser feito de diversas formas supervisionada, não-supervisionada, semi-supervisionada ou aprendizado ativo (RUSSEL e NORVIG, 2013).

As regras de classificação podem ser representadas computacionalmente como vetores e podem ser avaliadas por diversas medidas de qualidade como acurácia, precisão e sensitividade (*recall*) (ISHIDA, 2008). Elas podem ser avaliadas sistematicamente para que sejam encontradas a melhor combinação de atributos possível. Desta forma podemos resolver o AR com algoritmos de

otimização, visto que visa encontrar a melhor solução entre todas as viáveis, já que a cada iteração ele aceita e rejeita regras com base nas medidas selecionadas.

Pila (2007) propôs um algoritmo evolutivo para resolver o AR como um problema de otimização multi-objetivo utilizando duas medidas de qualidade de regras como função objetivo. Carvalho (2009) trabalhou com otimização por nuvem de partículas multi-objetivo (*Multiobjective Particle Swarm Optimization* – MOPSO) no aprendizado indutivo de regras e propôs algumas soluções de problemas na aplicação desta técnica no AR, sugerindo uma solução em múltiplas *threads* para acelerar a resolução do problema.

1.1 – MOTIVAÇÃO

As técnicas tradicionais de mineração de dados não são capazes de encontrar boas soluções para o *Big Data* em tempo hábil. Uma alternativa para trabalhar com essas grandes bases de dados é através da computação paralela. Em novembro de 2006 a NVIDIA apresentou o CUDA (*Compute Unified Device Architecture*), que é uma plataforma de computação paralela de propósito geral e um modelo que leva o motor de computação paralela das unidades de processamento gráfico (GPU). Esta plataforma foi pensada para resolver problemas complexos de forma mais eficiente que numa unidade de processamento central (CPU), visto que permite a execução do algoritmo na GPU fazendo uso de seus diversos núcleos para propósitos diversos que não sejam renderização de gráficos (NVIDIA, 2010).

Apesar do uso de técnicas sequenciais apresentar bons resultados em problemas de mineração de dados, há uma busca pelo desenvolvimento de algoritmos paralelos aplicados à MD. Estes algoritmos buscam reduzir o tempo de execução, especialmente quando é necessário tratar uma grande base de dados mantendo a qualidade do resultado gerado.

O algoritmo PSO é um algoritmo bioinspirado que se baseia no comportamento de enxames, nos quais alguns dos indivíduos são identificados como líder que guiam o enxame em busca das melhores soluções, e assim é implementado de forma a imitar a “inteligência de enxame” encontrada na natureza.

Na busca de algoritmos mais rápidos para a MD vários pesquisadores têm mostrado seus resultados. Böhm et al. (2009) utilizaram a plataforma CUDA para criar um algoritmo de MD utilizando placas gráficas (GPU) para executarem o algoritmo. Manavski e Valle (2008) utilizaram GPU para propor uma solução rápida para o algoritmo Smith-Waterman, para encontrar similaridades em bancos de dados de proteínas e DNA. Souza *et al.* (2011) fizeram uma implementação de *Particle Swarm Optimization* (PSO) utilizando a arquitetura CUDA, de forma a acelerar o algoritmo em problemas com muitos dados, o PSO-GPU de forma a ser adaptável a qualquer problema que possa ser resolvido usando PSO e conseguiram acelerar o processo em até 5 vezes.

No contexto do Aprendizado de Regras, Meneses (2016) trabalhou num algoritmo de Mineração de Dados que se utiliza de exploração de múltiplos enxames no aprendizado de regras de classificação não ordenadas. Ele explora um algoritmo multi-objetivo no qual as regras são aprendidas em apenas uma execução, não havendo preferência entre as regras e por isso criando um conjunto não ordenado de regras.

O algoritmo MOPSO-P proposto por Carvalho (2009) é uma variação multi-objetivo do PSO que divide o conjunto de treinamento em vários subconjuntos e executa a busca nesses subconjuntos em diversas threads. Meneses (2016) se aproveitou da arquitetura paralela das GPUs propôs uma implementação em CUDA do MOPSO-P. Este algoritmo recebeu o nome de M-PSO e obteve um aumento na velocidade de execução por trocar o processamento em série da CPU pelo processamento paralelo da GPU. Porém, não foi implementada a parte da divisão da base de dados, todos os enxames tinham acesso a toda base de dados durante a execução do algoritmo.

1.2 – OBJETIVOS

Partindo da hipótese de que a geração de um algoritmo que particiona a base de dados entre diversos núcleos de processamento irá reduzir o tempo de execução sem que haja perda na qualidade do algoritmo, este trabalho tem como objetivo geral desenvolver um algoritmo de otimização por enxame de partículas paralelo para o problema de aprendizado de regras. Como objetivos específicos desenvolver um algoritmo de otimização paralelo que compartilhe além da base

de dados os conjuntos de líderes entre todos os enxames, desenvolver um segundo algoritmo que mantenha os líderes separados e que particione a base de dados entre os enxames de forma a acelerar a execução e por fim investigar o desempenho destes algoritmos.

1.3 – METODOLOGIA

Para alcançar estes objetivos foi feita uma revisão bibliográfica para entendimento da Mineração de Dados e Aprendizado de Regras. Foi realizado estudo das especificidades utilizadas pela plataforma CUDA na sua implementação em C/C++. Foi realizado um estudo do algoritmo M-PSO para conhecer seus detalhes. Após isto foram desenvolvidos os dois algoritmos que são uma extensão do M-PSO, um com compartilhamento de líderes e um com divisão da base de dados por enxame, este último com a intenção de aprimorá-lo. Por último foram realizados experimentos para a validação do algoritmo otimizado.

Para a validação deste trabalho o M-PSO, e os gerados por este trabalho foram executados dez vezes para cada uma das sete bases de dados utilizadas, obtendo resultados de qualidade similar em todas as execuções, mas o algoritmo otimizado obteve um tempo de execução até 51,8% inferior ao algoritmo original e o algoritmo com líderes compartilhados obteve tempos similares ao original.

1.4 – ORGANIZAÇÃO DO TRABALHO

Este trabalho está dividido em cinco partes, a primeira é a Introdução, com noções gerais do trabalho. Na segunda parte são aprofundados conceitos de Mineração de Dados, regras de classificação, algoritmos de otimização e plataforma CUDA, e lista alguns trabalhos relacionados. A terceira parte descreve os algoritmos utilizados para o desenvolvimento deste trabalho. Na quarta parte são descritos os experimentos feitos para validação do algoritmo, além de apresentar os resultados e discussão. Na quinta e última parte são feitas as considerações finais.

2 – FUNDAMENTAÇÃO TEÓRICA

Neste capitulo são apresentados conceitos de Mineração de Dados, Bases de Dados, Aprendizado de Regras, Métricas de cálculo de qualidade de regras, Regras de Classificação. Também são apresentados conceitos de CUDA, como funciona a estrutura da GPU e comparação de seu funcionamento com a CPU, codificação em CUDA. São listados alguns trabalhos relacionados e por último apresentados conceitos e algoritmos que foram utilizados de base para o desenvolvimento deste trabalho.

2.1 – MINERAÇÃO DE DADOS

Segundo Han, et al. (2012) o processo de descobrimento do conhecimento (*Knowledge-discovery in databases* – KDD) segue os seguintes passos: Limpeza de Dados, Integração de Dados, Seleção de Dados, Transformação de Dados, Mineração de Dados, Avaliação de Padrões e Apresentação do Conhecimento. Os primeiros quatro passos são diferentes formas de processamento dos dados para a mineração propriamente dita, sendo assim a MD um passo no KDD, sendo o passo essencial pois ele descobre padrões escondidos para a avaliação. Apesar do KDD se referir a todo o processo de extração de informações de uma base de dados, a indústria e mídia tem usado o termo Mineração de Dados como referência a todo este processo e não somente a um dos passos.

Diversos padrões podem ser minerados, como:

- Caracterização, quando se usa sumarização de um atributo por uma característica de um ou mais atributos como por exemplo caracterizar um empregado por salário anual;
- Discriminação, quando se atribui um valor a um atributo no registro em função de outros;
- Associações, também conhecia como grupos de afinidade, objetiva determinar que “coisas” estão relacionadas, descobrindo suas regras de associação;
- Regras de classificação do tipo SE-ENTÃO, onde a presença de algum atributo num evento implica a presença de outro no mesmo evento;

- Regressão para análises preditivas, que busca uma função que relacione as variáveis para extrapolar dados futuros;
- Análises de agrupamento que segmenta o conjunto num número de subgrupos homogêneos, também conhecido como *clustering*.

Todos esses que podem ser minerados dependendo da necessidade ou foco da pesquisa.

2.1.1 – Base de Dados

Dado é um fato, um valor documentado ou um resultado de medição, que quando tem um sentido ou significado atrelado se torna informação, e quando esta informação pode ser utilizada para tomar decisões surge o conhecimento. (Silva, 2016). Bases de dados são um conjunto de arquivos relacionados entre si com registros sobre pessoas, lugares ou coisas. São coleções organizadas de dados que se relacionam de forma a criar algum sentido.

Cada dado em uma base de dados é descrito por um conjunto fixo de atributos: $A_i, i \in \{1, \dots, n_{atr}\}$. Um atributo pode assumir um conjunto finito de valores, um número inteiro (discreto) ou número real (contínuo). Por exemplo, seja e_j um vetor de valores de atributos rotulado com sua respectiva classe (atributo meta), $e_j = \left(v_{1,j}, \dots, \ v_{n_{atr},j}, c_{i,j}\right)$ no qual cada $v_{i,j}$ é um valor possível do atributo A_i e $c_{i,j}$ é uma dos n_{cl} possíveis valores do atributo classe. Um conjunto de cardinalidade n_{ex} é um conjunto contendo n_{ex} exemplos, como exemplificado na tabela 2.1.

Tabela 2.1 – Conjunto de exemplos no formato atributo-valor

	A_1	A_2		$A_{n_{atr}}$	C
e_1	$v_{1,1}$	$v_{2,1}$	$\cdots$	$v_{n_{atr},1}$	c_1
e_2	$v_{1,2}$	$v_{2,2}$	$\cdots$	$v_{n_{atr},2}$	c_2
$\vdots$	$\vdots$	$\vdots$	$\cdots$	$\vdots$	$\vdots$
$e_{n_{ex}}$	$v_{1,n_{ex}}$	$v_{2,n_{ex}}$	$\cdots$	$v_{n_{atr},n_{ex}}$	$c_{n_{ex}}$

A tabela 2.2 apresenta um exemplo de base dados sobre o tempo e prática de esportes. Os exemplos têm 4 atributos, Previsão (categórico), Temperatura

(continuo), Umidade (contínuo), Vento (discreto) e a Classe Praticar esportes (sim ou não).

Tabela 2.2 – Conjunto de informações sobre o tempo para a prática de esportes

Previsão	Temperatura	Umidade	Vento	Praticar Esportes
sol	30	85	não	não
sol	26	90	sim	não
nublado	28	86	não	sim
chuvoso	21	90	não	sim
chuvoso	20	80	não	sim
chuvoso	19	70	sim	não
nublado	18	65	sim	sim
sol	22	95	não	não
sol	20	70	não	sim
chuvoso	24	80	não	sim
sol	24	70	sim	sim
nublado	22	90	sim	sim
nublado	27	75	não	sim
chuvoso	21	91	sim	não

Para que esses dados já previamente organizados sejam minerados no processo de KDD eles podem passar por diversos tratamentos de pré-processamento como:

- Seleção de atributos, que ajuda a diminuir dimensionalidade selecionando as colunas mais relevantes da tabela que varia de acordo com a informação a ser minerada;
- Limpeza dos dados, que abrange qualquer tratamento para assegurar a qualidade dos dados;
- Discretização, a transformação de atributos contínuos em discretos;
- Binarização, dependendo do algoritmo pode ser necessário transformar atributos discretos e contínuos em um ou mais atributos binários;

- Transformação de variáveis, diversos métodos podem ser utilizados, mudança de escala, como por exemplo de -1 a 1 ou de 0 a 1. Isto pode se fazer necessário para evitar que alguns atributos influenciem mais que outros por terem escalas discrepantes.

2.1.2 – Aprendizado de Regras

A Mineração de Dados exige ferramentas inteligentes no auxílio da descoberta do conhecimento em grandes bases de dados. Nesta tarefa se usa muito de algoritmos de Aprendizado de Máquina (AM). O Aprendizado de Máquina é uma área da Inteligência Artificial que procura construir máquinas que podem melhorar seus resultados. No AM, o Aprendizado de Regras que objetiva construir regras se utiliza de dados rotulados, ou seja, possuem uma informação da classe.

O AR pode ser supervisionado, quando envolve aprendizagem a partir de exemplos das entradas e saídas, as regras geradas são avaliadas se representam a saída esperada. O aprendizado supervisionado tem por objetivo classificar novos exemplos dentro das classes conhecidas. O AR também pode ser não supervisionado, que aprende padrões a partir da entrada e não verifica as saídas, neste aprendizado as entradas não são rotuladas e o objetivo é descobrir similaridades entre elas.

O processo de analisar os relacionamentos existentes entre dados de uma base com objetivo de encontrar correlações ou associações é chamado de descoberta de regras de associação. (Silva, 2016)

Métodos baseados em regras são uma classe popular de técnicas em aprendizado de máquina e mineração de dados (FÜRNKRANZ, et al. 2012). Eles compartilham o objetivo de encontrar padrões nos dados de forma a serem expressos numa regra SE-ENTÃO e podem ser divididas em descobrimento de regras descritivas e aprendizado de regras preditivas, essas regras relacionam determinada condição a um determinado resultado. (HAN, et al. 2012).

O antecedente contém uma ou mais restrições de atributos e o consequente uma classe, no antecedente as restrições não precisam ser sobre todos os

atributos e para cada atributo apenas uma restrição. Podemos tirar um exemplo de regra da tabela 2.2:

SE Previsão = sol E umidade = 70 ENTÃO classe = sim

Esta notação de regra pode ser simplificada num par ordenado (consequente, antecedente) ou (corpo, cabeça), respectivamente e representado da seguinte forma:

$$corpo \rightarrow cabeça \text{ ou } B \rightarrow H\ (Body, Head)$$

Tomando um exemplo, e_i, da base de dados, uma regra *R* cobre o exemplo se todos os valores satisfazem as restrições da regra e cobre corretamente o exemplo se ambos são da mesma classe. Então, se *R* cobre e_i, *B* é verdadeiro. Se cobre corretamente *B* e *H* são verdadeiros.

2.1.2.1 – Matriz de Contingência

As regras precisam ser avaliadas individualmente para que formem um conjunto representativo e possa ter grande abrangência e assim tornar a análise mais precisa. Esta análise permite a descoberta de quais regras melhor representam o domínio.

Para medir a qualidade das regras se usa a matriz de contingência. Para melhor entendimento tomemos como exemplo a Tabela 2.3. Na Tabela 2.3 B representa o conjunto de exemplos para os quais o corpo de *R* é verdadeiro e $\bar{B}$ para os quais o corpo de *R* é falso. O mesmo funciona para H quando a cabeça da regra é verdadeira e $\bar{H}$ quando é falsa. Já b representa a cardinalidade de B, ou seja, o número de exemplos de corpo verdadeiro para *R*, assim como as demais $\bar{b}$, h e $\bar{h}$ representam as demais cardinalidades e n_{ex} representa o total de exemplos (que é dado pela soma $h + \bar{h}$ ou $b + \bar{b}$).

Tabela 2.3 – Matriz de Contingência

	H	$\bar{H}$	
B	hb	$\bar{h}b$	b
$\bar{B}$	$h\bar{b}$	$\overline{hb}$	$\bar{b}$
	h	$\bar{h}$	n_{ex}

2.1.2.1.1 – Medidas de Avaliação

A partir dos valores de matriz de contingência é possível calcular diversas medidas para as regras, medidas estas que são definidas através de probabilidade condicional, e estas são as mais usadas:

- Precisão: Mede o quanto a regra é específica para o problema. Quanto maior o valor, mais a regra está associada à classe. Também conhecida como confiança.

$$Precisão = P(H|B) = \frac{P(HB)}{P(B)} = \frac{hb}{b}$$

- Erro: Complemento da precisão, quanto maior o erro menos a regra está associada à classe

$$Erro = P(\bar{H}|B) = \frac{\bar{h}b}{b}$$

- Confiança negativa: É o equivalente à precisão, mas para exemplos não cobertos pela regra.

$$Confiança\ Negativa = P(\bar{H}|\bar{B}) = \frac{\overline{hb}}{\bar{b}}$$

- Precisão de Laplace: Equivalente à Precisão, mas penaliza regras que cobrem poucos exemplos. N_{cl} representa o número de classes do domínio

$$Precisão\ de\ Laplace = P(H|B) = \frac{P(HB)}{P(B)} = \frac{hb + 1}{b + N_{cl}}$$

- Sensitividade: Medida relativa dos exemplos da classe positiva que são cobertos. Também conhecida como completeza

$$Sensitividade = P(B|H) = \frac{hb}{h}$$

- Especificidade: equivalente da sensitividade para exemplos não cobertos. Quanto maior o valor, menos exemplos da classe negativa são cobertos de forma errônea.

$$Especificidade = P(\bar{B}|\bar{H}) = \frac{\overline{hb}}{\bar{h}}$$

- Cobertura: Medida relativa ao número de exemplos cobertos, exemplos em que o corpo é verdadeiro sem considerar a cabeça.

$$Cobertura = P(B) = \frac{b}{n_{ex}}$$

- Suporte: é a medida relativa do número de exemplos cobertos corretamente.

$$Suporte = P(HB) = \frac{hb}{n_{ex}}$$

2.1.2.2 – Regras de Classificação

Quando se fixa um dos atributos da base de dados como consequente (classe) e utiliza-se os outros como possíveis antecedentes, tomando como exemplo a Tabela 2.2 onde o atributo “Praticar Esportes” foi usado como consequente, pode-se criar regras que associam os valores de atributos antecedentes a um valor do atributo classe que foi escolhido como alvo, a essas regras chamamos regras de classificação.

Para garantir a qualidade das regras de classificação são feitas duas etapas no algoritmo o treino e o teste, geralmente sorteia-se aleatoriamente cerca de dez porcento da base de dados para servir como teste e o restante como treino. Na fase de treino o algoritmo busca os padrões de atributos antecedentes e suas associações com as classes gerando as regras. Na fase de teste os casos sorteados são utilizados com as regras geradas e a saída das regras é confrontada com a classe dos casos teste, caso a saída seja idêntica à classe a regra acertou e contabilizamos um acerto, caso não contabilizamos um erro, esses erros e acertos são utilizados para construir a Matriz Confusão.

2.1.2.2.1 – Matriz Confusão

É uma matriz onde se organizam as informações para as regras, ela é utilizada para medir a taxa de erros na classificação através de exemplos ainda não conhecidos e é construída após a indução do classificador.

Mostrando todos os exemplos utilizados no aprendizado quais eram positivos e classificados corretamente (*True Positive* – TP), quais eram negativos e foram classificados corretamente (*True Negative* – TN), quais eram positivos

classificados como negativos (*False Negative* – FN) e quais eram negativos e classificados como positivos (*False Positive* – FP). Como na Tabela 2.4.

Tabela 2.4 – Exemplo de Matriz Confusão

Dados / **Classificação**	**POSITIVO**	**NEGATIVO**
POSITIVO	TP	FN
NEGATIVO	FP	TN

2.1.2.2.2 – Medidas de Avaliação

A partir dos valores da Matriz Confusão podemos retirar as seguintes medidas:

- Taxa de erro por classe: calculada utilizando o total classificado incorretamente para classe dividido pelo total na classe, por exemplo número de classificados corretamente como positivos dividido por todos os positivos do conjunto de dados.

$$Taxa\ de\ Erro\ (+) = \frac{FP}{TP + FN}$$

$$Taxa\ de\ Erro\ (-) = \frac{FN}{TN + FP}$$

- Taxa de erro total: calculada utilizando o total de amostras classificadas erradas divididas pelo total de amostras.

$$Erro\ Total = \frac{FP + FN}{N}$$

- Acurácia: calculada utilizando o total de amostras classificadas corretamente dividida pelo total de amostras. Quanto maior o valor desta medida, maior são os acertos do classificador.

$$Acurácia = \frac{TP + TN}{N}$$

- *Recall*: também conhecida como sensitividade, é calculada dividindo o total de positivos classificados corretamente dividido pelo total de positivos. Quanto maior esta medida, maiores são os acertos para uma determinada classe. Como por exemplo o *recall* para classe positiva.

$$Recall = \frac{TP}{TP + FN}$$

- Precisão: calculada dividindo o total de positivos classificados corretamente pelo total que foi classificado como positivo. Quanto maior a precisão, menor é o número de erros de classificação.

$$Precisão = \frac{TP}{TP + FP}$$

- Especificidade: calculada dividindo o total que foi classificado corretamente como negativo pelo total de amostras negativas. Quanto maior o valor desta medida, menos exemplos da classe negativa são cobertos de forma errônea pela regra.

$$Especificidade = \frac{TN}{TN + FP}$$

- *F-Measure*: relaciona os valores da precisão e do recall. É útil por medir o *trade-off* entre precisão e recall. Calculada segundo a fórmula abaixo.

$$F - measure = \frac{2 * Precisão * Recall}{Precisão + Recall}$$

2.1.2.3 –Aprendizado de regras como problema de otimização

Pela definição de AR o programa busca aprender um conceito a partir de um conjunto de exemplos. As regras induzidas são uma combinação de valores dos atributos e um valor da classe. E para isto é necessário codificar as regras em vetores. Tomando por exemplo a Tabela 2.2 que tem como atributos: Tempo, Temperatura, Umidade, Vento, PraticarEsportes (classe: sim ou não). Extraindo uma regra dela:

Regra: SE Previsão = sol E umidade = menor ou igual a 70 ENTÃO classe = sim

Podemos codificar esta regra como uma partícula : <sol;?;menor ou igual a 70;?;sim> e ela deve ser codificada de forma numérica, uma possível codificação pode ser: “sol” = 1 (Tempo), “menor ou igual a 70” = 1 (umidade), “sim” = (classe PraticarEsportes). Os valores vazios representados como 0. Assim o exemplo acima pode ser codificado como: <1;0;1;0;1>

A indução das regras se dá através de três passos: Estado inicial onde as regras são inicializadas aleatoriamente ou qualquer outra forma de inicialização, Função Objetivo que verifica se as regras produzem os resultados esperados e Função Sucessor que define quais ações devem ser executadas para as mudanças de estado nas regras que é feita alterando os valores para atributos incluindo a classe. A partir desses passos um algoritmo de busca recebe como entrada um ou mais estados iniciais e executa um conjunto de ações para obter regras que atinjam os objetivos.

Considerando o AR como um problema de busca, um critério de avaliação precisa ser definido para decidir qual conjunto de regras é a solução esperada para um dado problema. Enumerar todas as regras possíveis é ineficiente, custoso e para uma base de dados grande pode ser completamente inviável. É um problema combinatório e diversas técnicas podem ser utilizadas, entre elas as mais comuns usam abordagem da cobertura e algoritmos gulosos.

A estratégia dos algoritmos tradicionais para AR é executada iterativamente. A cada iteração o algoritmo busca, encontra a melhor regra e remove exemplos não cobertos por ela. E repete-se este procedimento até que todos os exemplos sejam cobertos.

Mas existem outros algoritmos que fazem outras abordagens nesta área. Eles são baseados em Metaheurísticas Multiobjetivo (MHMO). Nesses algoritmos as propriedades das regras podem ser avaliadas através de objetivos diversos podendo construir regras mais simples e mais precisas (TORACIO e POZO, 2007). Mas a construção de modelos multiobjetivos não é trivial (ISHIDA, 2008). Nestas estratégias as regras podem ser encontradas em uma única execução. As técnicas MHMO permitem a criação de classificadores compostos

por regras com propriedades específicas que exploram os conceitos da dominância de Pareto (TORACIO e POZO, 2007).

Na área de otimização multiobjetivo a noção de solução ótima é diferente, pois é desejado que se atinja um equilíbrio entre os diversos objetivos ao invés de um objetivo atingir um ponto ótimo. Como há conflito entre os objetivos o acréscimo no valor de um deles é sempre ligado ao decréscimo em outro. E por isso normalmente não há somente uma melhor solução e sim um conjunto que representa o melhor compromisso entre os objetivos. (REYES-SIERRA e COELLO, 2006).

Um problema de AR tem como objetivo construir regras de conhecimento com propriedades específicas, que são relacionadas a medidas de avaliação das regras. Pode-se usar apenas uma métrica, que seria um problema de objetivo simples, mas quando procura-se otimizar diversas métricas tem-se um problema multi-objetivo.

Em 1995 Kennedy e Eberhart propuseram um método para otimização de funções não lineares contínuas que simulava comportamento social de uma população de agentes como um enxame, baseados em princípios básicos de inteligência de enxames, a Otimização por Nuvem de Partículas (*Particle Swarm Optimization* – PSO), que se mostrou uma forma extremamente simples e efetiva para otimização de uma grande quantidade de funções.

No PSO o conjunto de possíveis soluções é um conjunto de partículas que é denominada enxame ou população, e elas se movem no espaço de busca. Cada partícula representa uma possível solução que é uma posição no espaço de estados. Desta forma cada conjunto de soluções para uma regra pode ser definida como uma partícula, e a busca pela melhor posição da partícula, a busca pela regra que melhor representa o conjunto de dados.

Ainda temos o MOPSO (*Multiple Objective Particle Swarm Optimization*) que é um algoritmo derivado do PSO, que trabalha no mesmo sentido de representação de partículas e movimentação do espaço das soluções. Mas ele trabalha com múltiplos objetivos. E da mesma forma que outros MHMO podem ser utilizados para resolver o AR o MOPSO pode ser utilizado.

2.2 – CUDA

CUDA é uma plataforma de computação paralela de propósito geral, que inclui modelo de programação e um conjunto de instruções que permite o uso do poder de computação das GPUs da NVIDIA. Ela foi anunciada em 2006 e foi pensada para resolver problemas complexos de forma mais eficiente que numa CPU. Nesta plataforma pode se fazer uso dos diversos núcleos da GPU para propósitos diversos que não sejam renderização de gráficos, podendo executar tarefas com alto nível de paralelismo.

Computação paralela é uma forma de computação onde várias operações são realizadas simultaneamente, ela pode ser utilizada em qualquer problema que possa ser dividido em partes menores independentes entre si, no caso específico das placas gráficas, esta computação realiza exatamente a mesma operação em todos os núcleos, mas com diferentes dados em cada núcleo, possibilitando dessa forma acelerar muitas vezes qualquer algoritmo que tire proveito dessas propriedades.

Na documentação da CUDA o computador na qual a placa de vídeo está instalada é denominado *host* e a placa de vídeo onde são executados as *threads* paralelas é denominado *device*. Esta denominação é utilizada nas funções para indicar onde elas são executadas. Todo algoritmo em CUDA precisa que uma parte seja executada no *host*, pois todo computador trabalha primariamente na CPU, e posteriormente dados são transferidos para o *device* para a execução de funções paralelas.

As GPUs tem uma capacidade de computação de ponto-flutuante muito maior que as CPUs, pois elas são especializadas em computação intensiva e altamente paralela. A renderização de gráficos tem essas exigências, e por isso tem mais transístores dedicados ao processamento de dados que o de cache e controle de dados como ilustrado na Figura 2.1.

A Figura 2.1 mostra a estrutura de uma CPU e uma GPU, comparando as áreas de transistores, separando em Controle, Cache, Unidades Lógico-Aritmética (ALU) e Memória Cache e DRAM. Na CPU temos uma maior quantidade de transístores alocados para controle e cache e menos para ALU. Na GPU a maior parte está alocada nas ALU, que é onde ocorrem as operações,

com menor importância para o controle e cache, visto que o controle é feito por bloco de execução. Os blocos executam a mesma operação em diversos núcleos simultaneamente com dados diferentes.

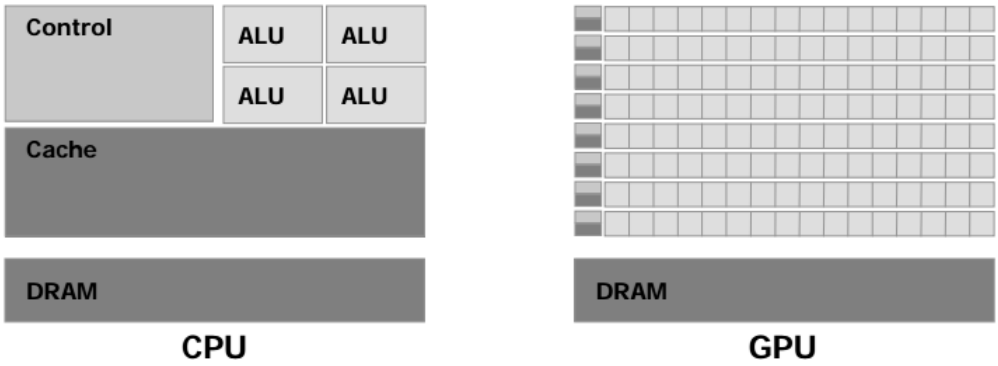

Figura 2.1 – GPU reserva mais transístores para processamento de dados

As GPUs são mais apropriadas para resolver problemas que podem ser expressados como computações paralelas, o mesmo programa sendo executado em vários elementos de dados, com alta intensidade aritmética. Como o mesmo programa é executado para cada elemento de dados, há pouco requerimento de um controle de fluxo sofisticado. Nesse modelo o algoritmo é executado em vários elementos com alta intensidade aritmética, então a latência de acesso a memória pode ser ocupada com cálculos ao invés de grandes caches de dados (NVIDIA, 2010).

A plataforma CUDA vem com um ambiente que permite o uso de C como linguagem de alto nível, mas aceita outras linguagens como FORTRAN, DirectCompute, OpenCL, OpenACC, Java e Python.

O modelo de programação escalável em CUDA permite distribuir a execução do programa de acordo com as possibilidades de cada GPU, desde GPUs profissionais como as séries Quadro e Tesla como as voltadas para público gamer de alto ou baixo custo da linha GeForce. Cada placa é dotada de uma matriz com diferentes quantidades de Multiprocessadores de Streaming (SM), e os blocos do programa são alocados de acordo com a GPU, cada SM empgrega uma arquitetura chamada SIMT (*Single Instruction, Multiple Threads* – do inglês Instrução única, Múltiplos *Threads*).

A arquitetura SIMT é ilustrada na Figura 2.2, mostrando como o mesmo programa CUDA se comporta em duas GPUs diferentes. Na primeira com dois

SM e a segunda com quatro SM. Um conjunto de blocos de execução é executado simultaneamente em todos os SM, após isso outro conjunto de blocos, até finalizarem todos os blocos do programa. Cada bloco representa uma *thread*.

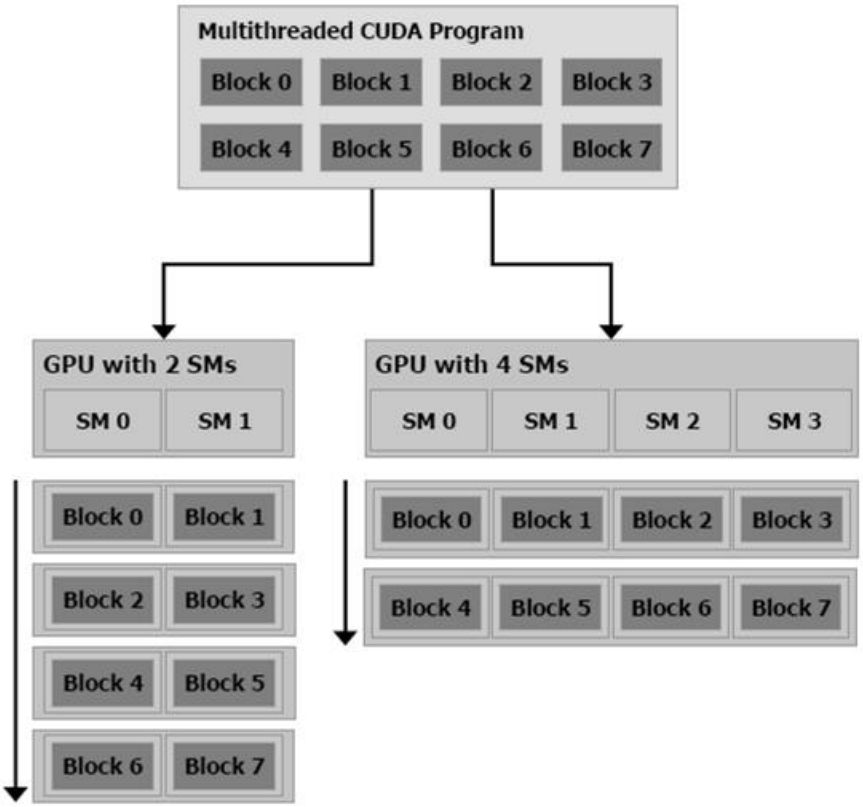

Figura 2.2 – O mesmo programa em CUDA executado em GPUs com diferentes números de Multiprocessadores de Streaming

Um código CUDA em C é muito similar a um código em C com algumas chamadas específicas da plataforma como pode ser observado no *"Hello World!"* exemplificado no Quadro 2.1. Inicialmente é criada uma função *hello* que realiza a soma de dois vetores em paralelo, então o *host* imprime na tela o vetor que contém a palavra *Hello*, na sequencia os vetores auxiliares são alocados na memória do *device*, os vetores contendo números a serem somados e o que contém *Hello* são copiados para o *device* é chamada a função que realiza a soma dos vetores, o resultado da soma é coletado para a variável que originalmente continha a *Hello* e que agora contem *World!* , libera a memória do *device*, e finalmente imprime na tela *World!* finalizando o código.

```
#include <stdio.h>

const int N = 16;
const int blocksize = 16;

__global__ void hello(char *a, int *b){
     a[threadIdx.x] += b[threadIdx.x];
}

int main(){
     char a[N] = "Hello \0\0\0\0\0\0";
     int b[N] = {15, 10, 6, 0, -11, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0};

     char *ad;
     int *bd;
     const int csize = N*sizeof(char);
     const int isize = N*sizeof(int);

     printf("%s", a);

     cudaMalloc( (void**)&ad, csize );
     cudaMalloc( (void**)&bd, isize );
     cudaMemcpy( ad, a, csize, cudaMemcpyHostToDevice );
     cudaMemcpy( bd, b, isize, cudaMemcpyHostToDevice );

     dim3 dimBlock( blocksize, 1 );
     dim3 dimGrid( 1, 1 );
     hello<<<dimGrid, dimBlock>>>(ad, bd);
     cudaMemcpy( a, ad, csize, cudaMemcpyDeviceToHost );
     cudaFree( ad );
     cudaFree( bd );

     printf("%s\n", a);
     return 0;
}
```

Quadro 2.1 – Código “Hello World!” utilizando a plataforma CUDA

2.3 – TRABALHOS RELACIONADOS

Pila (2007) propôs um algoritmo evolutivo para construir regras de conhecimento com propriedades específicas de forma isolada, sem considerar a interação entre regras, que utiliza uma estrutura de representação de regras que

possibilita considerar uma grande quantidade de operadores evolutivos usando uma função multi-objetivo que considera mais que uma medida de avaliação de regra.

Manavski e Valle (2008) utilizaram GPU para propor uma solução rápida para o algoritmo Smith-Waterman, para encontrar similaridades em bancos de dados de proteínas e DNA. Böhm et al. (2009) utilizaram a plataforma CUDA para propor vários algoritmos para tarefas computacionalmente caras de MD projetados para os ambientes altamente paralelos das GPU.

Carvalho (2009) trabalhou com *Multiobjective Particle Swarm Optimization* (MOPSO), que é uma MHMO, no aprendizado indutivo de regras e propôs algumas soluções de problemas na aplicação desta técnica no AR. E por final sugere o algoritmo que denominou MOPSO-P que tenta gerar um bom conjunto de regras utilizando atributos numéricos e atributos nominais, e tem por principal objetivo a execução múltipla do MOPSO, dividindo o treinamento em *p* partições e executando as diversas *threads* em paralelo.

Souza *et al.* (2011) fizeram uma implementação de PSO utilizando a arquitetura CUDA, de forma a acelerar o algoritmo em problemas com muitos dados, fizeram o PSO-GPU de forma a ser adaptável a qualquer problema que possa ser resolvido usando PSO e conseguiram acelerar o processo em até 5 vezes.

Yu et al. (2015) se utilizaram do Hadoop que é uma plataforma de computação distribuída para propor o algoritmo *Cloudoop*. Esse algoritmo é uma implementação do Hadoop para fazer agrupamento baseado em densidade distribuído de forma eficiente para big data. Propuseram inicialmente um algoritmo *CluC* para otimização de particionamento de células em torno de um ponto e é utilizado pelo *Cloudoop* de forma paralela.

Harris et al. (2016) trabalharam com otimização de parâmetros para aprendizagem de regras de classificação acelerada por GPU, usando um método baseado em busca em *grids* e validação cruzada que pode ser massivamente paralelizável e obtiveram resultados 28x mais rápidos utilizando GPU comparando com uma implementação de CPU com multi-thread.

Meneses (2016) utilizou como base o algoritmo MOPSO-P gerado por Carvalho (2009) que faz uso de múltiplas *threads* para execução do algoritmo MOPSO e implementou utilizando CUDA onde essas *threads* podem rodar de forma realmente paralela e obteve um ganho de performance quando comparado com o mesmo algoritmo rodando de forma não paralela.

2.4 – ALGORITMOS

Os algoritmos utilizados para este trabalho são PSO, MOPSO, MPSO-P e M-PSO, para entende-los é necessário estabelecer uma terminologia que segue nas definições abaixo:

- Partícula: Indivíduo do enxame, cada partícula representa uma solução potencial para o problema, a posição da partícula é determinada pela solução que ela representa no momento.

- Enxame: Grupo de partículas

- Dominância: Quando uma solução é avaliada como superior a outra é dito que uma solução domina a outra, quando o a qualidade avaliada é equivalente é dito que não há dominância entre as soluções

- pbest *(Particle best*): Melhor posição de uma dada partícula.

- gbest (*Global best*): Melhor posição de todas as partículas do enxame.

- Líder: partícula que é usada como guia para as outras

- Velocidade: Vetor que dirige o processo de otimização, ele determina a direção na qual a partícula deve se mover para melhorar a posição atual.

2.4.1 – PSO

O PSO se baseia num enxame de partículas que cooperam para encontrar a melhor solução global, elas possuem um vetor de posição e um vetor de velocidade, e essas grandezas são atualizadas a cada iteração do PSO, de acordo com as equações 3.1 e 3.2.

$$V_i[j] \leftarrow V_i[j] + c_1 r_1[j](pbest_i[j] - X_i[j]) + c_1 r_1[j](gbest[j] - X_i[j]) \quad (3.1)$$

$$X_{i+1}[j] \leftarrow X_i[j] + V_{i+1}[j] \quad (3.2)$$

Nas equações acima, $V_i[j]$ representa a velocidade da partícula *i* num vetor de *j* dimensões, $X_i[j]$ a posição da partícula *i* num vetor de *j* dimensões, $pbest_i[j]$ representa a dimensão *j* da melhor posição encontrada pela partícula i (melhor local), $gbest[j]$ representa a dimensão *j* da melhor posição encontrada por todas as partículas (melhor global), c_1 e c_2 são dois parâmetros que representam respectivamente o peso da influência do melhor local e melhor global para a nova velocidade, $r_1[j]$ e $r_2[j]$ são dois parâmetros distribuídos aleatoriamente entre [0,1].

O MOPSO é um derivado do PSO que tem múltiplos objetivos e por isso ao invés de buscar uma única solução ele possui um conjunto de soluções, um conjunto de melhores globais (gbest). Para gerar este conjunto cria-se um arquivo, que é uma estrutura de dados que armazena as melhores soluções encontradas até então e é atualizado a cada iteração. Para a atualização é feito o cálculo de velocidade da partícula, cálculo da nova posição, verificação se a posição encontrada na iteração atual não é dominada por nenhuma posição do arquivo, seleção de um líder a ser seguido no arquivo, e as posições no arquivo também são atualizadas caso sejam dominadas por uma nova posição, no final do processo temos um conjunto de posições solução (REYES-SIERRA e COELLO 2006).

O MOPSO além de implementar todos os métodos do PSO, deve implementar um método para manter e gerenciar o arquivo externo, chamado de método de arquivamento. Além disto deve implementar um método de escolha de líder, visto que cada enxame tem seu líder

No problema do AR cada regra pode ser representada por um vetor contendo seus atributos e classe, ela pode ser representada pela posição da partícula do PSO que também é um vetor. Cada valor categórico dos atributos das regras é mapeado para um valor numérico, desta forma todos os cálculos do PSO podem ser feitos com os atributos. Com isto podemos transpor um problema de AR para ser resolvido usando o PSO.

2.4.2 – MOPSO-P

Carvalho (2009) propôs o MOPSO-P que faz uso do modelo de múltiplos enxames para executar em múltiplas *threads* utilizando a técnica de dividir para conquistar diminuindo a complexidade do problema convertendo ele em buscas menores. Neste algoritmo, apresentado no Algoritmo 3.1, a base de dados é dividida em *p* partições, e em cada partição é executada o algoritmo MOPSO de forma independente, isolados em suas *threads*. Após esta execução paralela, as soluções são unidas, é realizado o cálculo do conjunto de líderes e a limpeza das soluções dominadas.

Algoritomo 2.1 – Algoritmo MOPSO-P	
1:	Divide a base de dados de busca em p partições
2:	Para cada um dos p enxames
3:	Executa o MOPSO
4:	Finaliza laço
5:	Une soluções dos enxames
6:	Calcula o conjunto de Líderes do conjunto de Soluções
7:	Remove soluções dominadas

Após a execução destes passos, as regras são testadas com a partição de testes da base de dados para avaliação da qualidade dos resultados obtidos.

2.4.3 – M-PSO

Meneses (2016) utilizou como base o MOPSO-P (Carvalho, 2009) para gerar o M-PSO, que é basicamente a implementação em CUDA deste algoritmo. O mapeamento foi feito de forma que cada enxame fosse mapeado a um bloco e cada partícula a um núcleo da GPU, maximizando desta forma o paralelismo entre as *threads* visto que eles rodam o mesmo algoritmo com dados independentes ao mesmo tempo.

Como no algoritmo MOPSO-P cada partícula é inicializada com um conjunto de regras aleatório, sendo esta sua posição inicial, assim como as velocidades são iniciadas aleatórias. A melhor posição local também recebe uma regra aleatória atribuída à posição atual. A melhor posição global não é inicializada. A velocidade de cada partícula é inicializada aleatoriamente. Após esta inicialização calcula-se o líder do enxame e calculada a *crowding distance* (CD) de cada partícula para cada objetivo escolhido.

Em seguida cada partícula recebe uma regra do conjunto de líderes recém calculado que representa melhor a posição global. Após isto o arquivo de soluções de cada enxame é preenchido com as regras atualizando as posições, velocidades, pbest, gbest, CD. Depois o conjunto de líderes é recalculado considerando as novas posições. Ao final as soluções de todos os enxames são coletadas e reunidas, o líder deste conjunto é calculado e as soluções não dominadas mantidas.

Mas durante a implementação do M-PSO, algumas dificuldades foram encontradas e a divisão das bases de dados de forma a aumentar a diversidade de soluções e reduzir o tempo de execução não foi implementada, desta forma o algoritmo final permitia que todos os enxames tivessem acesso a toda a base de dados de forma indistinta. E obteve um sucesso parcial, houve um grande ganho no tempo de execução de CPU *versus* GPU, mas não houve a implementação total do algoritmo MOPSO-P. Assim abrindo caminho para intervenções e melhorias a partir deste algoritmo e assim diminuir ainda mais o tempo de execução e aumentar a diversidade de soluções encontradas.

3 – ALGORITMOS

Para este trabalho foram desenvolvidos dois algoritmos chamados de M-PSO-LU e M-PSO-OT, o primeiro foi jeito para fins de comparação e validação do segundo, fazendo pequenas alterações no código original do M-PSO para que o arquivo de líderes fosse compartilhado, já no segundo foram feitas otimizações de forma a diminuir o tempo de execução do M-PSO e implementando funções que não haviam sido implementadas.

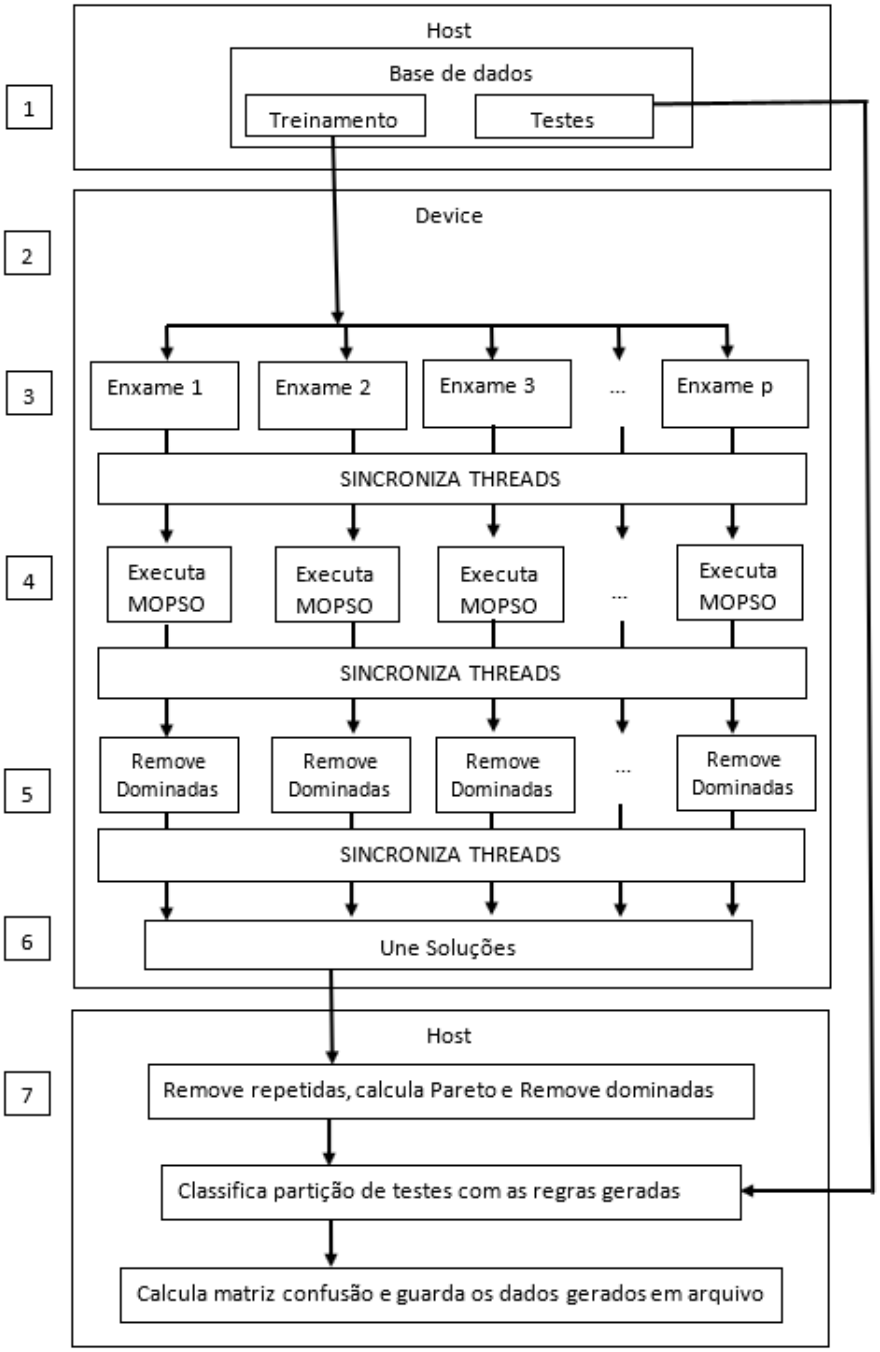

Figura 3.1 – Diagrama da execução do algoritmo M-PSO sem modificações

A figura 3.1 mostra um diagrama genérico de execução do M-PSO, o qual foi alterado de forma criar os algoritmos deste trabalho, nele é resumido todo o processamento e locais de execução seja no *host* ou no *device*. O *host* carrega na memória a base de dados já previamente dividida em suas partições e carrega a partição de treinamento para o *device*.

O Algoritmo 3.1 mostra o funcionamento básico do kernel em CUDA que foi utilizado para as implementações feitas. No *device* são inicializadas as partículas e todas suas informações iniciais. Após isso as *threads* são sincronizados, o algoritmo do MOPSO é executado para cada enxame, após isso as soluções dominadas são removidas, as soluções unidas e os dados ficam disponíveis para o *host*.

Com os dados no *host* as soluções repetidas são removidas, é feito um novo cálculo de dominância e removidas as soluções dominadas, são realizados os testes, a parte da base que foi separada para os testes é processada utilizando as regras descobertas pelo algoritmo e a matriz confusão é calculada. Por fim as regras e matrizes confusão das dez partições testadas são armazenadas em arquivo.

Algoritomo 3.1 – Funcionamento do kernel implementado em CUDA

```
1:  Inicializa a partícula
2:  Calcula matriz de contingência e funções objetivo da partícula
3:  Insere referência na memória compartilhada de seu enxame (bloco)
4:  Sincroniza execução do kernel da partícula
5:  Se a partícula (thread) for a primeira do enxame (bloco):
6:     Calcula conjunto de líderes das partículas
7:     Atualiza arquivo de soluções
8:     Atualiza Gbest das partículas
9:  Encerra bloco condicional
10: Sincroniza execução do kernel da partícula
11: Inicia laço
12:    Atualiza velocidade da partícula
13:    Atualiza posição da partícula
14:    Calcula objetivo da partícula
15:    Atualiza pbest da partícula
16:    Se a partícula for a primeira do enxame
17:       Calcula conjunto de líderes das partículas do enxame
18:       Atualiza arquivo de soluções
19:       Atualiza gbest das partículas
20:    Encerra bloco condicional
21: Finaliza laço
22: Preenche vetor de soluções finais
```

3.1 – M-PSO-LU

Este algoritmo foi gerado para validação do algoritmo final, o M-PSO-OT, ele se baseia no M-PSO feito por Meneses (2016), onde todos os enxames têm acesso a toda a base de dados, mas o líder é calculado de forma separada entre os enxames. No M-PSO-LU o conjunto de líderes é uma única variável compartilhada por todos os enxames, desta forma eles tem acesso a toda a base de dados e todos os valores de conjunto de líderes antes da união das soluções.

No trabalho original do M-PSO foi sugerido fazer uma comunicação entre os enxames, esse compartilhamento do conjunto de líderes é uma forma efetiva de comunicação entre os enxames. Porém desta forma o algoritmo poderia permitir uma convergência mais rápida, mas poderia perder qualidade de resultado, pois tenderia a uniformizar mais as soluções e perder em diversidade de regras geradas.

3.2.1 – Detalhes de Implementação

A partir da Figura 3.1 a principal modificação se dá no ponto 3, onde os enxames são separados, neste algoritmo todos os enxames têm acesso irrestrito à base de dados e os líderes são compartilhados de forma a todos os enxames terem acesso a todos os líderes, que são as primeiras *threads* de cada bloco. No código ao invés de iniciar uma nova variável de líderes para cada *thread* é iniciada uma variável compartilhada entre todas as *threads*.

A parte do *host* é idêntica ao algoritmo base, já no *device* é inicializado um único arquivo que vai conter os líderes de todas as partículas e suas soluções, e os enxames que vão executar individualmente em seus respectivos blocos *kernel* do M-PSO.

3.2 – M-PSO-OT

O algoritmo previamente gerado por Meneses (2016) foi utilizado como base para a geração do M-PSO-OT. Este algoritmo é uma extensão do M-PSO, onde os dados alocados na GPU foram separados de forma que cada enxame tivesse acesso a apenas uma partição da base.

Para evitar vieses a base de dados foi inicialmente embaralhada no *host* e posteriormente enviada para o *device*. Já com os dados no *device* a base de dados foi dividida de forma que todos os enxames recebessem a divisão inteira de dados pelo número de enxames, e o ultimo enxame recebia este valor acrescido do resto desta divisão inteira, os resultados eram reunidos e os melhores resultados eleitos apenas no final da execução.

3.2.1 – Detalhes de Implementação

A partir do algoritmo esquematizado na Figura 3.1 este algoritmo seguiu com algumas modificações. No passo 1 algoritmo faz o pré-carregamento das bases de dados para a memória principal do *host* e faz a aleatorização destes dados através de um algoritmo de embaralhamento de vetores mostrado no Quadro 3.1. Então é feita a criação dos enxames e inicialização das partículas, então os dados são copiados para o *device*.

```
int elementos_restantes = quant_exemp;
while (elementos_restantes>0){
   int k = rand() % (elementos_restantes);
   exemplo tmp = exemplos_h[k];
   exemplos_h[k] = exemplos_h[elementos_restantes-1];
   exemplos_h[elementos_restantes-1] = tmp;
   elementos_restantes--;
}
```

Quadro 3.1 – Embaralhamento do vetor de exemplos no *host*

Com os dados no *device* o *host* dispara a execução do *kernel* do M-PSO-OT. Neste ponto temos outra alteração do código base, no passo 2 da Figura 3.1 já com a base de dados copiada para o *device*, o acesso a ela para cada partícula é calculado usando aritmética de vetores, visto que a memória é compartilhada e a cache de cada bloco é pequena, dependendo do modelo do *device* tem poucas centenas de *Kilobytes*, perto do tamanho das bases cuja maior partição utilizada chegou a quase 5 *Megabytes*. A posição e tamanho dos vetores parciais utilizados é calculada usando o código mostrado no Quadro 3.2.

```
    int id = blockIdx.x * blockDim.x + threadIdx.x;

    if (id > quant_particulas * quant_enxames) return;


    int div, resto, valor_vect, quant_vect;

    div = quant_exemp / quant_enxames;

    resto = quant_exemp % quant_enxames;

    valor_vect = id >= (div*quant_enxames) ? ((id - resto) / (quant_enxames))
: (id / quant_enxames);

    quant_vect = div*quant_particulas;

    quant_vect += id >= (div*quant_enxames) ? resto : 0;
```

Quadro 3.2 – Cálculo de tamanho e posição de vetores parciais

Após a divisão da base é executado o algoritmo do MOPSO para todos os enxames, e as soluções eram sempre armazenadas na primeira partícula do enxame (primeira *thread* do bloco), também chamada de Líder.

Como na plataforma CUDA cada thread executa sempre o mesmo código que todas as outras o algoritmo original do MOPSO precisou de algumas modificações para que cada partícula fosse mapeada em uma *thread* e cada enxame mapeado para um bloco. O algoritmo 3.1 foi modificado de forma que antes da inicialização das partículas o acesso a cada uma delas fosse calculado de acordo com sua identificação, calculo este mostrado no Quadro 3.2.

Como pode-se visualizar no algoritmo 3.1, sempre que os dados dos enxames precisam ser analisados é necessário fazer sincronização das *threads*, que é um recurso da plataforma CUDA que bloqueia as *threads* até o final da execução, fazendo a chamada da função *__syncthreads()* , o código desta *kernel* pode ser visualizada de forma resumida no Quadro 3.3.

Após a execução do *kernel* da aplicação o *host* recebe os dados processados pelo *device* e inicia um pós-processamento. Como os enxames executam independentemente existe o risco de regras idênticas terem sido geradas, por isso elas devem ser apagadas e mantidas apenas uma cópia delas. No *device* o conjunto de líderes é calculada independentemente para cada

enxame, por isso a dominância precisa ser recalculada agora com todas as regras reunidas, e após isso as regras dominadas são removidas.

```
    int id = blockIdx.x * blockDim.x + threadIdx.x;
    if (id > (*param).quant_particulas * (*param).quant_enxames) return;
    extern __shared__ int mem[];
    particula *enxame_shared = (particula*)&mem[0];
    // Calcula posição e tamanho das partições do vetor
    particula p = enxames[id];
    //inicializa a partícula
    __syncthreads();
    regra* regras_enxame;
    regiao_pareto pareto;
    if (threadIdx.x == 0){ // calcula líderes, atualiza soluções e gbest }
    __syncthreads();
    regra* total;
    int i, quant_regras;
    for (i = 0; i < (*param).bl_interacoes; i++) {
    // atualiza velocidade e posição da partícula
    // calcula objetivo da particula
    //atualiza pbest
       __syncthreads();
       if (threadIdx.x == 0){ //se a partícula for a líder, calcula conjunto
de líderes, atualiza soluções e gbest }
       __syncthreads();
    }
    if (threadIdx.x == 0){ device_removeDominadas(&pareto); }
    __syncthreads();
    posicoesFinais[id] = enxame_shared[threadIdx.x].posicao;
    }
```

Quadro 3.3 – Código resumido da função kernel implementada em CUDA

Após esse pós-processamento das regras são realizados os testes, a parte da base que foi separada para os testes é processada utilizando as regras descobertas pelo algoritmo. Neste passo cada exemplo passa por todas as

regras que votam numa das duas classes, positiva ou negativa, para o exemplo em teste e em caso de empate uma classe é escolhida aleatoriamente. As classes votadas são comparadas com as classes reais disponíveis na base de dados para o cálculo da matriz confusão. Por fim as regras e matrizes confusão das dez partições testadas são armazenadas em arquivo.

3.3 – COMPARAÇÃO

A partir dos algoritmos base para geração deste trabalho e dos algoritmos criados pode-se fazer uma comparação entre eles, comparando o número de objetivos, divisão da BD, forma de execução e se há comunicação entre enxames, resumido na tabela 3.1.

Tabela 3.1 – Comparação entre algoritmos base e criados

Algoritmo	Objetivo	BD	Execução	Comunicação
PSO	Simples	Unificado	Sequencial (CPU)	Enxame único
MOPSO-P	Múltiplos	Dividido	Sequencial (CPU)	Não há
M-PSO	Múltiplos	Unificado	Paralela (CUDA)	Não há
M-PSO-LU	Múltiplos	Unificado	Paralela (CUDA)	Líderes compartilhados
M-PSO-OT	Múltiplos	Dividido	Paralela (CUDA)	Não há

4 – EXPERIMENTOS

Neste capítulo são detalhados os experimentos executados para a validação dos algoritmos gerados na execução do presente trabalho, indicando qual metodologia, *device*, bases de dados e parâmetros utilizados, assim como os resultados obtidos destes experimentos e discussão.

4.1 – METODOLOGIA

A alteração e compilação do código foi feita utilizando o ambiente de desenvolvimento integrado (IDE) Microsoft Visual Studio 2015 no sistema Operacional Windows 10 para a compilação de código C utilizando a plataforma CUDA.

Os algoritmos M-PSO (original), o M-PSO-LU e M-PSO-OT foram executados num *device* GeForce GTX 1060 com 6GB de RAM, 1280 núcleos CUDA e frequência de 1570MHz.

4.1.1 – Bases de Dados

Para a execução e análise da performance dos algoritmos, utilizamos sete bases de dados do mundo real, elas estão disponíveis no *UCI Machine Learning Repository* (Lichman, 2013), e estão descritas na tabela 4.1, os atributos originais reais ou inteiros foram discretizados, pois o algoritmo utilizado cria regras a partir de atributos discretos, usando o software WEKA (FRANK *et al.,* 2016) forçando a divisão em 4 faixas de valores com número aproximadamente igual de amostras para não haver viés para uma faixa com grande parte da população amostral.

Tabela 4.1 – Bases de dados utilizadas nos experimentos

Nomes	**Atributos**	**Exemplos**
breast	10	699
bupa	7	345
glass	10	214
satimage	37	180000
kr-vs-kp	37	57910
lettera	17	28760
vehicle	19	7610

Estas bases de dados foram selecionadas buscando variar o número de atributos e exemplos. Como não haveria tempo para executar apenas em grandes bases de dados foram privilegiadas bases com características diversas entre si para submeter o algoritmo a bases mais variadas e observar a variação de comportamento. Devido a isto apenas satimage e kr-vs-kp possuem um grande número de exemplos.

O Software WEKA (FRANK *et al.,* 2016) também foi utilizado também para o particionamento das bases de dados nas 10 partições de testes utilizadas

4.1.2 – Algoritmos e Parâmetros

Para os experimentos foram utilizados três algoritmos, M-PSO, M-PSO-LU e M-PSO-OT, que fizeram a mineração de dados nas sete bases escolhidas.

Para cada execução definiu-se que cada algoritmo realizaria 25 interações com uma população total de 40 partículas que foram distribuídas em 10 enxames, sendo que cada um deles poderia armazenar até 2 soluções. Além disso, foi estabelecido que seriam geradas regras para as classes positiva e negativa e que as funções objetivo consideradas seriam a sensitividade e a especificidade. Estes parâmetros foram selecionados por possibilitar o teste do executável na máquina onde ele foi desenvolvido, um *device* GeForce 635M com 2GB de RAM, 96 núcleos CUDA e frequência de 475MHz

As execuções foram realizadas utilizando-se a técnica de validação cruzada (cross-validation) com dez partições. Dessa forma, cada base de dados foi dividida em dez partições de mesmo tamanho, sendo nove partições utilizadas para o treinamento do algoritmo e uma partição escondida da base e utilizada somente para a posterior realização do teste. Sendo assim, os algoritmos foram executados sobre cada base dez vezes. Ao longo de cada execução, alterou-se a partição utilizada para o teste (cada partição só poderia ser utilizada para testes apenas uma vez).

Cada algoritmo foi executado 10 vezes, matrizes de confusão obtidas para cada execução foram somadas e o calculada a média dos tempos para comparação dos resultados obtidos.

Foi gerado um script em Python 3.6 que automatizou as alterações no arquivo de configuração para modificar as bases, e rodar todas as vezes necessárias cada um dos algoritmos com cada uma das bases de dados, somar os valores das matrizes confusão e dos tempos, realizar os cálculos das métricas escolhidas e gerar um relatório com os resultados.

A partir dos dados compilados pelo script foram utilizados para a comparação o tempo de execução e a qualidade das regras geradas em termos de classificação, utilizando as métricas escolhidas.

4.1.3 – Medidas

Para uma comparação de qualidade dos resultados obtidos na execução dos algoritmos para cada base de dados foram utilizadas as matrizes confusão obtidas para calcular as métricas: acurácia por dar a precisão total e *f-measure* que permite avaliar precisão e *recall* ao mesmo tempo.

4.2 – RESULTADOS E DISCUSSÃO

Após a execução dos algoritmos e recolhidas as saídas, as matrizes de confusão e os tempos de execução de todas as execuções foram somadas para cada algoritmo e para cada base de dados individualmente, dados que podem ser observados na tabela 4.2 que contém os tempos totais de execução, as diferenças dos algoritimos modificados com relação ao original e o ganho de tempo em porcentagem.

Tabela 4.2 –Tempo de execução para os três algoritmos por base de dados

Algoritmo	**Tempo médio (ms)**	**Diferença**	**Ganho (%)**
	breast		
M-PSO	675,16		
M-PSO-LU	683,26	-8,10	-1,20
M-PSO-OT	536,65	138,51	20,52
	bupa		
M-PSO	423,87		
M-PSO-LU	449,67	-25,81	-6,09
M-PSO-OT	301,05	122,81	28,97
	glass		
M-PSO	449,36		
M-PSO-LU	461,76	-12,40	-2,76
M-PSO-OT	360,89	88,47	19,69
	satimage		
M-PSO	4067,73		

M-PSO-LU	4110,18	-42,44	-1,04
M-PSO-OT	2203,12	1864,61	45,84
kr-vs-kp			
M-PSO	2357,22		
M-PSO-LU	2383,84	-26,63	-1,13
M-PSO-OT	1442,83	914,39	38,79
lettera			
M-PSO	5886,45		
M-PSO-LU	6005,05	-118,60	-2,01
M-PSO-OT	2837,46	3048,99	51,80
vehicle			
M-PSO	996,00		
M-PSO-LU	990,71	5,29	0,53
M-PSO-OT	795,56	200,44	20,12

Meneses (2016) comparou a execução do algoritmo em CPU e GPU obtendo um ganho de até 52,74% no tempo médio de execução do algoritmo apenas na mudança de plataforma onde o algoritmo era executado, e com essa modificação pode-se adicionar um ganho ainda maior sobre o algoritmo rodando na CPU.

Observando a tabela 4.2 podemos ver a diferença de tempo na execução do algoritmo M-PSO-OT em relação ao algoritmo M-PSO, obtendo sempre um ganho de tempo como é mostrado de forma relativa na tabela 4.2 e que varia de 19,69% a 51,80%, mostrando assim uma melhor performance do algoritmo proposto por este trabalho, sendo que os maiores ganhos foram nas bases de dados com maior número de exemplos. Enquanto o M-PSO-LU se comportou com tempo entre similar (0,53% mais rápido) e um pouco pior (6,09% mais lento).

A partir das matrizes confusão foram calculadas a acurácia, e *f-measure* para comparação de qualidade entre os algoritmos. Os resultados encontram-se descritos na tabela 4.3.

Tabela 4.3 – Métricas calculadas para os três algoritmos por base de dados

Algoritmo	**Acurácia**	***F-measure***	**Ganho tempo (%)**
breast			
M-PSO	0,65	0,55	
M-PSO-LU	0,63	0,56	-1,20
M-PSO-OT	0,64	0,55	20,52
Bupa			
M-PSO	0,54	0,50	
M-PSO-LU	0,56	0,53	-6,09
M-PSO-OT	0,56	0,51	28,97
Glass			
M-PSO	0,59	0,17	
M-PSO-LU	0,64	0,18	-2,76

M-PSO-OT	0,57	0,18	19,69
	Satimage		
M-PSO	0,51	0,16	
M-PSO-LU	0,51	0,16	-1,04
M-PSO-OT	0,51	0,17	45,84
	kr-vs-kp		
M-PSO	0,52	0,51	
M-PSO-LU	0,53	0,52	-1,13
M-PSO-OT	0,53	0,51	38,79
	Lettera		
M-PSO	0,55	0,08	
M-PSO-LU	0,58	0,08	-2,01
M-PSO-OT	0,58	0,08	51,80
	Vehicle		
M-PSO	0,54	0,35	
M-PSO-LU	0,55	0,35	0,53
M-PSO-OT	0,56	0,34	20,12

Comparando estes resultados com os obtidos por Meneses (2016) que ao mudar a execução do algoritmo de CPU pra GPU e obteve uma diferença máxima de 4,63% a menos na *recall* do algoritmo rodado na GPU. Neste trabalho. Comparando as métricas entre os algoritmos M-PSO e M-PSO-OT as diferenças foram bem pequenas, variando entre -0,02 e 0,02 na Acurácia e entre -0,01 a 0,03 no *F-measure*, mantendo assim uma qualidade de resultados similar e com um considerável ganho no tempo de execução.

Com base nestes dados pode-se constatar que o algoritmo M-PSO-OT gerado para este trabalho conseguiu realizar a mesma tarefa do M-PSO com diferenças muito pequenas na qualidade, e um ganho de tempo de 19,69% até 51.80%, ou seja uma das bases foi processada em menos da metade do tempo, e em todas houve um ganho de tempo considerável. O M-PSO-OT representa um avanço sobre o M-PSO pois realiza a mesma tarefa num tempo reduzido.

5 – CONSIDERAÇÕES FINAIS

O estudo da tecnologia CUDA dá uma nova visão sobre os algoritmos computacionais, devido a possibilidade de executar uma grande quantidade de núcleos que permitem várias *threads* simultâneas. O uso de GPU na mineração de dados tem um grande potencial para diminuir o tempo de processamento das bases de dados, e os ganhos aparentam ser maiores em bases de dados maiores.

Neste trabalho foi gerado um algoritmo de otimização por enxames de partículas utilizando-se de múltiplos enxames para geração de regras de classificação. A base de dados foi dividida entre os diversos enxames com intuito de acelerar a execução.

Esta divisão da base de dados entre os diversos enxames proporcionou um ganho considerável de tempo, mas devido ao tamanho das bases e da cache não foi possível separar nas memórias dos blocos. Propõe-se que o algoritmo possa ser otimizado ainda mais de forma a decompor as bases de dados a um ponto que caibam inteiramente na memória cache dos blocos e permitir assim um maior ganho de velocidade de processamento.

Além disto o algoritmo básico ainda precisa de melhorias com relação a qualidade dos resultados gerados, observando os resultados obtidos percebe-se que há um espaço para melhora, visto que a acurácia tem um valor máximo de 1 (100% de acerto), e com isso sugere-se um estudo para aumentar a qualidade das soluções do M-PSO-OT que mantenha a velocidade de obtenção destas.

6 – REFERÊNCIAS BIBLIOGRÁFICAS

BÖHM , C., NOLL, R., PLANT, C., WACKERSREUTHER, B., ZHERDIN, A. Data Mining Using Graphics Processing Units. Transactions on Large-Scale Data and Knowledge-Centered Systems I, vol. 5740 of Lecture Notes in Computer Science, pp. 63–90, Springer, Berlin, Germany, 2009.

CARVALHO, A. B., Otimização por nuvem de partículas multiobjetivo no aprendizado indutivo de regras: Extensões e Aplicações. Dissertação de mestrado, Universidade Federal do Paraná, 2009.

CAVANILLAS, J.M., CURRY, E., WAHLSTER, W., New Horizons for a Data-Driven Economy: A Roadmap for Usage and Exploitation of Big Data in Europe. Editora Springer Open 2015.

FRANK, E., HALL, M. A., WITTEN, I. H. The WEKA Workbench. Online Appendix for "Data Mining: Practical Machine Learning Tools and Techniques", Morgan Kaufmann, Quarta Edição, 2016.

FACEBOOK, State of Connectivity 2015, A Report on Global Internet Access, 2016. disponível em <https://fbnewsroomus.files.wordpress.com/2016/02/state-of-connectivity-2015-2016-02-21-final.pdf> acesso em 05 de novembro de 2016.

FÜRNKRANZ, J., GAMBERGER, D., LAVRAC, N. Foundations of Rule Learning. Springer-Verlag 2012

HARRIS, G., PANANGADAN, A.M PRASANNA, V.K., GPU-Accelerated Parameter Optimization for Classification Rule Learning, Proceedings of the Twenty-Ninth International Florida Artificial Intelligence Research Society Conference, 2016.

HAN, J., KAMBER, M., JIAN, P., Data Mining: Concepts and Techniques. Third Edition. Waltham, USA: Morgan Kaufmann Publishers, 2012.

ISHIDA, C. Explorando Abordagens Inovadoras para Geração de Classificadores, Tese de Doutorado, Universidade Federal do Paraná, 2008.

KENNEDY, J., EBERHART, R., Particle Swarm Optimization. IEEE International Conference on Neural Networks, p 1942 – 1948. IEEE Press, 1995.

LICHMAN, M. UCI Machine Learning Repository [http://archive.ics.uci.edu/ml]. Irvine, CA: University of California, School of Information and Computer Science, 2013.

MANAVSKI, S., VALLE, G., Cuda compatible gpu cards as eficiente hardware accelerators for Smith-waterman sequence aligment. BMC Bioinformatics 9, 2008.

MENESES, M. C. C., Explorando Múltiplos Enxames no Aprendizado de Regras de Classificação Não-Ordenadas. Relatório de IC, Universidade Federal de Sergipe, 2016.

NVIDIA, C. U. D. A. Programming guide. 2010.

PILA, A. D., Computação Evolutiva para a construção de regras de conhecimento com propriedades específicas. Tese de Doutorado. Universidade de São Paulo, 2007.

PRATI, R. C., Novas abordagens em aprendizado de máquina para a geração de regras, classes desbalanceadas e ordenação de casos. Tese de Doutorado. Universidade de São Paulo, 2006.

REYES-SIERRA, M., COELLO, C.A.C., Multi-objective particle swarm optimizers: A survey of the state-of-the-art. Internacional Journal of Computational Intelligence Research, 2(3), 2006.

RUSSEL, S., NORVIG, P., Inteligência Artificial, Terceira Edição, Rio de Janeiro, Editora Elsevier, 2013.

SILVA, L. A., PERES, S. M., BOSCARIOLI, C., Introdução à mineração de dados com aplicações em R, 1ª Edição, Rio de Janeiro, Editora Elsevier, 2016.

SOUZA, D.L., MONTEIRO, D. M., MARTINZ, T.C., DIMITRIEV, V.A., TEIXEIRA, O. N. . PSO-GPU: accelerating particle swarm optimization in CUDA-based graphics processing units. Proceedings of the 13th annual conference companion on Genetic and evolutionary computation. ACM, 2011.

TORACIO, A., POZO, A., Multiple Objective particle swarm for classification-rule discovery. Proceedings of CEC 2007, p. 684-691. IEEE Computer Society, 2007.

TORACIO, A. Aprendizado de Regras de classificação com otimização por nuvem de partículas multiobjetivo. Dissertação de mestrado, Universidade Federal do Paraná, 2008.

TURNER, V., GANTZ, J. F., REINSEL, D., & MINTON, S. The digital universe of opportunities: rich data and the increasing value of the internet of things. Rep. from IDC EMC. 2014, disponível em <http://www.emc.com/leadership/digital-universe/2014iview/executive-summary.htm> acesso em 6 de novembro de 2016.

YU, Y. ZHAO, J., WANG, X. WANG, Q., ZHANG, Y. Coudoop: An Efficient Distributed Density-Based Clustering for Big Data Using Hadoop. International Journal of Distributed Sensor Networks, 2015.