UNIVERSIDADE ESTADUAL DE PONTA GROSSA
SETOR DE CIÊNCIAS AGRÁRIAS E DE TECNOLOGIA
DEPARTAMENTO DE INFORMÁTICA

William Xavier Maukoski

ALGORITMO K-MEANS EM AMBIENTE MANYCORE PARA REDUÇÃO DO TEMPO DE RESPOSTA DA MINERAÇÃO DE DADOS

PONTA GROSSA
2019

William Xavier Maukoski

ALGORITMO K-MEANS EM AMBIENTE MANYCORE PARA REDUÇÃO DO TEMPO DE RESPOSTA DA MINERAÇÃO DE DADOS

Dissertação apresentada para obtenção do título de Mestre em Computação Aplicada na Universidade Estadual de Ponta Grossa, Área de concentração: Computa- ção para Tecnologias Agrícolas.

Orientador: Prof. Dr. Luciano José Senger

PONTAROSSA
2019

Maukoski, William Xavier
M449 Algoritmo k-means em ambiente manycore para redução do tempo de resposta da mineração de dados / William Xavier Maukoski. Ponta Grossa, 2019.
78 f.

Dissertação (Mestrado em Computação Aplicada - Área de Concentração: Computação para Tecnologias em Agricultura), Universidade Estadual de Ponta Grossa.

Orientador: Prof. Dr. Luciano Jose Senger.

1. Computação paralela. 2. Mineração de dados. 3. Manycore. 4. Agricultura de precisão. I. Senger, Luciano Jose. II. Universidade Estadual de Ponta Grossa. Computação para Tecnologias em Agricultura. III.T.

CDD:004

Ficha catalográfica elaborada por Maria Luzia Fernandes Bertholino dos Santos- CRB9/986

TERMO DE APROVAÇÃO

William Xavier Maukoski

ALGORITMO K-MEANS EM AMBIENTE MANYCORE PARA REDUÇÃO DO TEMPO DE RESPOSTA DA MINERAÇÃO DE DADOS

Dissertação aprovada como requisito parcial para obtenção do grau de Mestre no Programa de Pós-Graduação em Computação Aplicada da Universidade Estadual de Ponta Grossa, pela seguinte banca examinadora:

Prof. Dr. Luciano José Senger

UEPG - Presidente

Prof. Dr. Renato Porfírio Ishii

UFMS

Prof. Dr. Arion de Campos Junior

UEPG

Ponta Grossa, 29 de maio de 2019.

AGRADECIMENTOS

Primeiramente agradeço A D’us pelo dom da vida e porque até aqui suA Mão tem me sustentado.

Agradeço ao meu orientador, o professor Dr. Luciano Jose Senger, pelo seu apoio e dedicação ao orientar-me, a sua disponiblidade e assiduidade em auxiliar-me, e seu esforço, sendo ele de grande importancia para a conclusão deste trabalho.

Agradeço aos meus colegas, alunos do programa de computação aplicada, pela convivencia e experiencias academicas compartilhadas. Em especial ao meu colega Henike Guilherme Jordan Voss, pelos trabalhos em dupla nas discplinas.

Agradeço aos docentes do Programa de Computação Aplicada da Universidade Estadual de Ponta Grossa, pelas aulas ministradas e pela consultoria em diversos momentos de duvidas para a execuçãodo do presente trabalho, em especial aos professores Maria Salete Marcondes Vaz e Rafael Mazer Etto, pelas oportunidades academicas que proporcionaram-me.

Agradeço a Universidade Estadual de Ponta Grossa, pela infraestrutura fornecida para realizar minha pesquisa.

Agradeço a Capes, pelo auxilio social para poder dedicar-me exclusivamente a pesquisa.

"O que sabemos é uma gota, o que ignoramos é um Oceano."
(Sir Isaac Newton)

RESUMO

A mineração de dados (MD) é potencialmente onerosa, estudos para diminuir o tempo de resposta são essenciais para conseguir entregar resultados em tempos menores. Muitas soluções propostas por trabalhos correlatos utilizam computação em clusters (aglomerados) de computadores, uma alternativa a isto é utilizar a computação com GPU. A programação com GPU, necessita um conhecimento aprofundado do algoritmo a ser trabalhado e de um entendimento da arquitetura da GPU que será utilizada. Este trabalho tem por objetivo geral investigar o uso da computação paralela em ambiente *manycore* para reduzir o tempo de resposta de algirtmos de MD. O algoritmo K-means por ser comumente adotados em tarefas de IA e sua característica NP-Difícil foi o escolhido para ser paralelizado. Foram utilizadas ferramentas para identificar o ponto de gargalo do K-means, simultaneamente foram feitas as avaliações dos pontos positivos e negativos destas ferramentas. Após identificar o ponto de gargalo do algoritmo, ele foi reescrito para ser executado com suporte da GPU, foram feitas as coletas dos tempos de respostas e pôr fim a medição do ganho de desempenho utilizando a GPU. Ao utilizar a GPU foi primeiro constatado um *SpeedUP* máximo de 7,09 e uma eficiência de 0,65% considerados pequenos ao comparar com outros trabalhos da literatura. Para contornar isto foi feito um aumento na base de dados utilizada aumentado o tempo de execução, assim obteve-se resultados melhores com um *SpeedUp* de 26,001 e uma eficiência de 2,4% ao utilizar o máximo de *cores* da GPU. Concluindo-se que é possível diminuir o tempo de resposta de algoritmos de mineração de dados utilizando GPU, sem precisar alterar o hardware do equipamento.

Palavras-chave: Computação Paralela, Mineração de Dados, *ManyCore*, Agricultura de Precisão

ABSTRACT

Data mining (MD) is potentially costly, studies to decrease response time are essential to deliver results in shorter times. Many solutions proposed by related work use computation in clusters of computers, an alternative to this is to use GPU computing. Programming with GPU requires a thorough knowledge of the algorithm to be worked on and an understanding of the GPU architecture that will be used. This work has the general objective to investigate the use of parallel computing in the manycore environment to reduce the response time of MD algorithms. The K-means algorithm for being commonly adopted in AI tasks and its NP-Difficult feature was chosen to be paralyzed. Tools were used to identify the bottleneck of K-means, while evaluating the positives and negatives of these tools. After identifying the bottleneck of the algorithm, it was rewritten to run with GPU support, collected response times, and ended the performance gain measurement using the GPU. When using the GPU, a maximum speed of 7.09 and an efficiency of 0.65% was found to be small when compared to other studies in the literature. To circumvent this was done an increase in the database utilized by increasing the run time, thus obtaining better results with a speed up of 26.001 and a efficiency of 2.4% when using the maximum of *colors* of the GPU. It is concluded that it is possible to decrease the response time of data mining algorithms using GPU, without having to change the hardware of the equipment.

Keywords: Parallel Computing, Data Mining, ManyCore, Precision Agriculture

LISTA DE FIGURAS

LISTA DE TABELAS

SUMÁRIO

1 INTRODUÇÃO

A preocupação com a escassez de produtos agrícolas remonta aos primórdios da humanidade. Tal preocupação tem sido objeto de estudo acadêmico desde o final do séc. XVIII e o desenvolvimento tecnológico e industrial diminuiu a perspectiva de falta de produtos agrícolas na atualidade. Porém, alguns problemas como aumentar a produtividade minimizando o impacto no meio ambiente, atender a logística para o transporte de produtos agrícolas, são ainda desafios para os pesquisadores.

Atualmente, há um incremento da presença de equipamentos e tecnologias para a aquisição, armazenamento e transmissão de dados na agricultura. Da mesma forma, há um aumento proporcional no volume de dados disponíveis. O aumento no volume de dados gera a necessidade de ferramentas capazes manipulá-los, além de extrair informação para auxiliar a tomada de decisão, esse crescente tecnológico é impulsionado pela rentabilidade econômica da agricultura, dentre as quais o maior destaque é da cultura da soja.

Segundo a Conab (Companhia Nacional de Abastecimento), em maio de 2018 houve uma produção de 116,996 milhões de toneladas de soja com uma produtividade de 3.333 kg/ha. O estado do Paraná é o segundo maior produtor de Soja do Brasil, sendo responsável por 19,070 milhões de toneladas de soja no mesmo período, com uma produtividade média de 3.503 kg/ha. O Brasil é o segundo maior produtor de soja do mundo sua produção é um inferior apenas a dos Estados Unidos da América, maior produtor mundial, sendo responsável pela produção de 119,518 milhões de toneladas[1].

Atualmente é uma importante ferramenta para auxiliar no incremento sustentavel de produtividade, a mineração de Dados (MD) é um conjunto de técnicas que fornece suporte à diversos campos da ciência, permitindo extrair padrões e relacionamentos, utilizando de algoritmos de Aprendizado de Máquina (AM). A partir do conhecimento obtido com a MD é possível predizer comportamentos e auxiliar na tomada de decisão (WITTEN; FRANK, 2015). Porém, a crescente escala de dados traz o desafio, de entregar essa predição em curto período de tempo. Uma das alternativas seria aumentar a capacidade de processamento do *hardware* utilizado (PÉREZ *et al.*, 2005), o que nem sempre é possível, devido à dificuldade em produzir um *hardware* mais potente.

Outro motivo do déficit de desempenho em algumas aplicações, é a demanda crescente de *software* pelo mercado, o que exige um aumento na produtividade dos programadores, causando um aumento na utilização de linguagens de alto nível. Resultando em códigos mais concisos, mas devido a natureza interpretada das linguagens

de alto nível demora-se mais para concluir tarefas (REN; AGRAWAL, 2011).

Sendo este um fenômeno real de programadores trocando o desempenho pela abstração na hora de programar. Mesmo utilizando sistemas de transliteração de código como o Shedskin[2], o qual traduz um código de Python para C++, possibilitando que compiladores de C++ possam ser usados para geração e código nativo, os resultados ainda são lentos (REN; AGRAWAL, 2011).

Algoritmos de mineração de dados muitas vezes são implementações de modelagens matemáticas e atualmente estes algoritmos encontram-se já implementados para serem utilizados por pesquisadores. Para facilitar a utilizaçãode de algoritmos de MD, tais algoritmos estão implementados em ferramentas com interface gráfica. Normalmente estas interfaces gráficas são desenvolvidas na mesma linguagem de programação que os algoritmos de MD foram implementados, é o caso da ferramenta Weka que teve seus algoritmos de MD todos desenvolvidos em Java, da mesma maneira que sua interface gráfica (HOLMES; DONKIN; WITTEN, 1994).

Java é uma linguagem popular em todos os seguimentos da computação, devido principalmente ao seu suporte a alta produtividade, e sua baixa curva de aprendizado. Sendo ela uma linguagem interpretada e não compilada, o que é claramente um grande fator para seu baixo desempenho. Além disto, um dos fatores mais atrativos do Java é seu rico suporte a estrutura de dados. Ao mesmo tempo que a escrita dinâmica do Java dá flexibilidade para programadores, também aumenta a sobrecarga do sistema computacional. Uma solução é utilizar as ferramentas da própria linguagem para programação multinúcleo nos processadores (REN; AGRAWAL, 2011).

A maioria dos trabalhos correlatos utiliza a arquitetura distribuída, como por exemplo aglomerados (PÉREZ *et al.*, 2005). Infelizmente as arquiteturas distribuídas envolvem um aumento considerável nos custos e podem ainda sofrer de problemas de transferência de data (SCHADT, 2010). O aumento do poder de processamento no próprio dispositivo utilizando GPUs, parece ser uma solução complementar oferecendo uma boa relação custo benefício por não precisar de uma reestruturação de *hardware*. (SCHADT, 2010).

Trabalhos na literatura propõem o aumento de desempenho utilizando a GPU. Estes trabalhos também propõem a utilização de interfaces ou de um *framework* o desenvolvimento com a GPU(KUMAR *et al.*, 2011). Porém, para que uma aplicação utilize todo o potencial dessa arquitetura é necessário escrever programas que fragmentem as coleções de dados do problema, trabalhem individualmente com esses fragmentos, unam o resultado do processamento desses fragmentos, sendo capazes de fornecer certa confiabilidade em relação aos resultados.

Desta forma, o objetivo geral deste trabalho é investigar o uso da computação paralela para reduzir o tempo de resposta da mineração de dados agrícolas, que sejam onerosos computacionalmente, ao utilizar ambiente *many core*. Seus objetivos específicos são: investigar o uso de arquiteturas de ambiente *many core* para melhorar o desempenho da mineração de dados agrícolas; obter indicações de qual a melhor alternativa para melhorar o tempo de resposta; verificar se o ganho no tempo de resposta pode ser obtido com soluções ao nível de *software* e não necessariamente com a melhoria no *hardware* e testar ferramentas de abstração para programação utilizando GPUs.

Conseguiu-se com este trabalho um *speed up* de 7,09 com uma eficiência de 0,65% em um primeiro momento, foi concluído que a base de dados era pequena, foi feito um incremento artificial no número de registros para realizar novos testes. Com a base de dados incrementada foi obtido um *speed up* de 26,001 com uma eficiencia de 2,4%.

1.1 ORGANIZAÇÃO DO TRABALHO

Este trabalho foi organizado da seguinte forma: No capítulo 1 é mostrado a importância da preocupação em atender a demanda de produtos agrícolas, a importância da soja para este setor e é abordado alguns fatores que limitam o desempenho de aplicações. Ainda neste capitulo são mostrados os objetivos gerais e específicos deste trabalho.

No capítulo 2 é mostrado a revisão da literatura, onde aborda o histórico da programação com GPU, novas tecnologias para abstrair a programação para GPU, mineração de dados, fundamentação teórica para medição de desempenho de soluções paralelas, e métodos para mensurar o ganho de desempenho.

No Capitulo 3 são descritos os materiais e métodos utilizados neste trabalho, suas escolhas e os motivos que os levaram a ser escolhidos.

No capítulo 4 são apresentados os resultados do trabalho, avaliação dos *profilers* utilizados, valores dos tempos de execução da versão com CPU e com GPU, os resultados das medidas de desempenho e discussão de todos estes dados. Neste capitulo consta também trechos referentes ao processo para paralelizar o algoritmo K-means.

No capítulo 5 são feitas as conclusões com base nos resultados do estudo e são descritos trabalhos futuros que podem ser realizados utilizando a versão do Weka produzido neste trabalho.

Este estudo conta ainda com o Apêndice A que detalha como configurar o

Windows para poder desenvolver novas funcionalidades para o Weka. No Apêndice B foi descrito como realizar a mesma configuração para o Linux, além de como configurar para o desenvolvimento com CUDAe integração das ferramentas usadas as instruções de linha de comando para poder utilizar o JIT corretamente. No Apêndice C está transcrito o método moveCentroidscom as alterações feitas neste estudo.

2 REVISÃO DA LITERATURA

2.1 AGRICULTURA, DO ESTUDO MALTHUSIANO À AGRI- CULTURA DE PRECISÃO

Em sua obra intitulado "*An Essay on the Principle of Population*", que foi publicado pela primeira vez em 1798, Thomas Malthus apresenta um estudo que argumenta a favor de uma escassez de alimentos decorrente da população crescente, tal estudo ficou famoso por estimar que a população cresceria em progressão geométrica, en- quanto a produção agrícola cresceria em uma progressão aritmética(MALTHUS, 1872). Atualmente a teoria malthusiana é dada como vencida por dois fatores principais:(i) A tendência de diminuir a população humana ao passo que os países tornam-se desenvolvidos (RICKLEFS, 2010). (ii) O aumento da produtividade de insumos agrícolas através de novas tecnologias e estudos (FOLEY *et al.*, 2005).

O fator (i) é algo recente na economia, sendo uma característica observada apenas no final do séc. XX inicio do séc. XXI (RICKLEFS, 2010). O fator (ii) também era impossível de ser previsto no período histórico do estudo de Malthus, pois ainda não havia iniciado o período conhecido como revolução industrial (1820), este foi um período de início da mecanização de trabalhos anteriormente manuais (ROBERT, 2004).O principal período de mecanização de trabalhos agrícolas é conhecido como revolução verde, momento compreendido entre 1945 a 1960 nos EUA e países Europeus, e posteriormente nos países emergentes, e em alguns países pobres não tendo ocorrido ainda (HAZELL, 2009).

Os avanços tecnológicos trouxeram grande crescimento para a agricultura nos últimos anos tendo vindo a aumentar progressivamente o rendimento do agronegócio. Aumentar a produção de alimentos de forma sustentável, utilizando a mesma área agriculturável e minimizando impactos ambientais gerados pelo uso e manejo de recursos agrícolas são algumas metas para a agricultura do século XXI (HERRERO *et al.*, 2010).

2.2 TAXINOMIA DE FLYNN

Um conceito teórico importante para escolher a abordagem que será empregada neste trabalho é a Taxonomia de Flynn, que é uma categorização de diferentes arquiteturas de computadores baseadas em como o computador interpreta um fluxo de instruções de dados, ela divide os computadores em quatro categorias, que serão

abordadas a seguir:

- **SISD** (*single instruction single data* - Instrução única e dado único, em tradução livre), nesta arquitetura existe uma única unidade de processamento, que executa o fluxo de instruções
- **SIMD** (*single instruction multiple data* - Instrução única e dados múltiplos, em tradução livre), nesta arquitetura uma única instrução opera sobre um múltiplo fluxo de dados.
- **MISD** (*multiple instruction single data* - Instruções múltiplas e dado único, em tradução livre), é um arquitetura hipotética.
- **MIMD** (*multiple instruction multiple data* - Instruções múltiplas e dados múltiplos, em tradução livre). Fluxos de instruções diferentes operam em fluxos de dados diferentes, podendo ser subdividas em duas arquiteturas distintas, arquitetura com memória compartilhada e arquitetura com memória distribuída.

Arquitetura com memória compartilhada são empregadas em aglomerados de computadores, do inglês *clusters*, podendo conter computadores homogêneos, ou não. Cada computador em um aglomerado é chamado de nó, eles podem estar fracos ou fortemente interligados através de *software*, podendo ser considerado, em alguns quesitos, um sistema único. Em um sistema com memória distribuída o trabalho é escalonado para ser processado no nó mais disponível, ou que o trabalho seja dividido e resolvido em mais de um núcleo do aglomerado (SENGER, 2005).

A computação paralela com memória distribuída é baseada em um computador com alto poder de processamento, sendo o mais comum os computadores em processadores robustos. No entanto, uma alternativa viável para obter alto poder de processamento é utilizar a GPU, que trazem uma nova perspectiva de alto poder de processamento devido a sua arquitetura responsável por realizar muitos cálculos.

2.3 PARALELISMO

Processadores originalmente foram desenvolvidos com um núcleo, mas com o aumento nos dados e funções que um computador precisa executar, criando uma demanda por processamento, sendo uma solução aumentar a velocidade de ciclo do processador, o que ocasionou um aumento no consumo de energia e na temperatura.

Estes fatores tornaram-se incentivos para o desenvolvimento da arquitetura multinúcleo, além do que utilizar um núcleo sem alteração na arquitetura, apenas replicando o número do mesmo, diminui a chance de falhas de *design*. Há, ainda,

fatores limitantes para um ganho efetivo no desempenho com apenas incremento da velocidade dos ciclos do processador, como por exemplo: a barreira da memória, o déficit entre a velocidade do processador e a memória RAM, e necessidade de aumentar a memória cache para mascarar a latência entre a memória RAM e o processador.

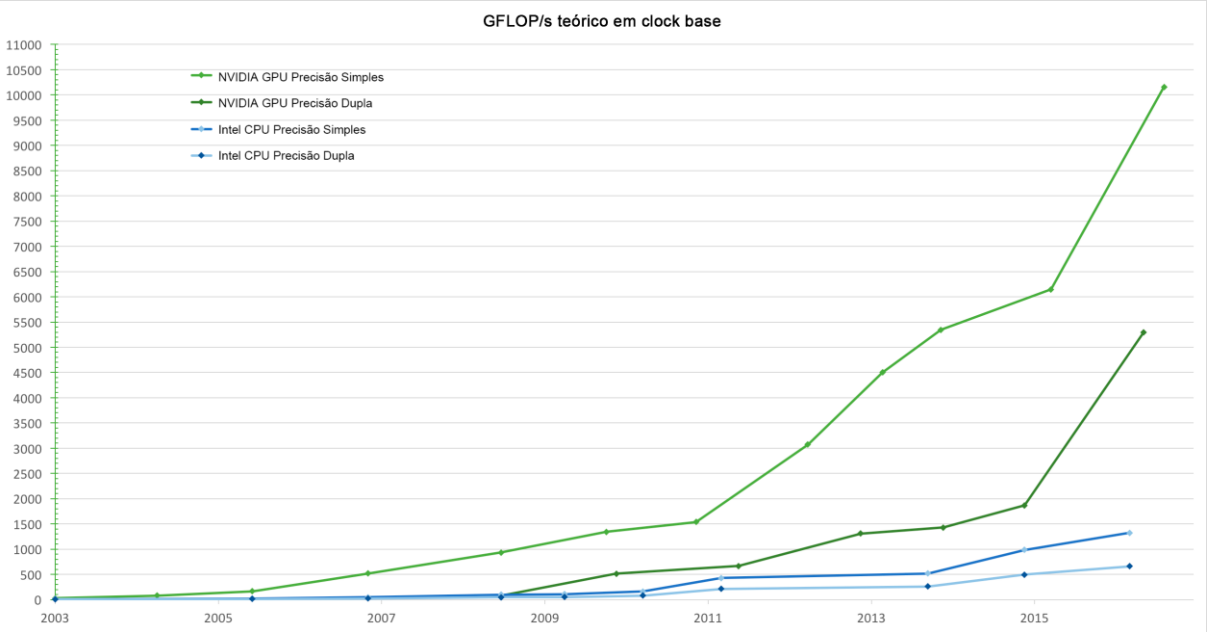
Também existem barreiras para o paralelismo, como o limite do quanto é possível paralelizar em uma única cadeia de instruções mantendo um desempenho razoável. A barreira da energia, que é o crescimento do consumo de energia e da temperatura ao utilizar *hardware* com arquitetura paralela.

Nos últimos anos, muito tem-se feito para mudar a perspectiva de processamento, saindo do paradigma sequencial e indo para o paradigma paralelo. Há 10 anos era comum ter *desktops* utilizando processadores com dois núcleos, atualmente (2018), existem processadores com 16 núcleos e 32*threads*, disponíveis para o usuário domestico[1]. Em ambientes de servidor, há processadores com 64 núcleos. Isto gera a necessidade de aplicações que possam utilizar todo o potencial dessa arquitetura.

Assim como a evolução dos processadores para desktops, nos últimos anos houve um incremento na utilização das GPUs (*Graphics Processing Unit* Unidade de Processamento Gráfico) como ferramenta de processamento de dados. As GPUs trabalham com uma arquitetura conhecida como *manycore*, devido ao seu número de núcleos maior do que as CPUs. Em 2019 existem GPUs domesticas com mais de 3000 nucleos[2]. Com isto fica clara a diferença da capacidade de processar tarefas simultaneamente entre uma CPU e uma GPU. Na figura 1 é mostrada a evolução da capacidade de processamento em Gigaflops (*Floating-point Operations Per Second*) de CPUs e de GPUs desde 2003 até 2016, evidenciando a diferença de poder de processamento entre as duas unidades ao longo dos ultimos anos.

[1] https://www.amd.com/pt/products/cpu/amd-ryzen-threadripper-1950x)
[2] https://www.geforce.com/hardware/desktop-GPUS/geforce-gtx-1080-ti/specifications

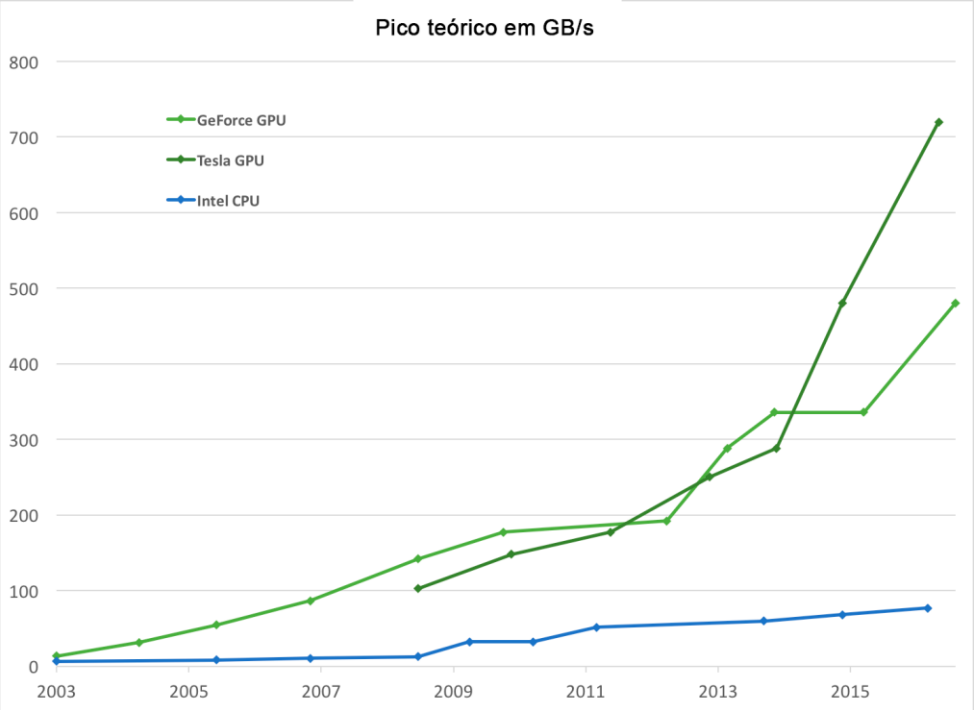
Figura 1 – Comparativo da capacidade Flops entre uma CPU e uma GPU



Fonte: CUDA *software* Stack (NVIDIA, 2019)

Por ser capaz de processar um fluxo de dados maior do que as CPUs é preciso que as GPUs tenham uma capacidade de banda (*bandwidth*, que é a capacidade a qual os componentes conseguem trasmitir dados) maior, como pode ser observado na figura 2.

Figura 2 – Comparativo da capacidade *bandwidth* teórico entre uma CPU e uma GPU



Fonte: CUDA *software* Stack (NVIDIA, 2019)

Programar utilizando a GPU exige que o desenvolvedor conheça a fundo o algoritmo que será paralelizado, em nível mais profundo do que apenas a documentação fornecida em alguns casos, precisando expender muito tempo do desenvolvimento estudando o algoritmo, fazendo depurações e testes de mesa. A programação com GPU é potencialmente complexa e exige um estudo aprofundado da arquitetura da GPUs utilizada precisando especializar-se nela, isto é um desafio que as próprias desenvolvedoras das GPUs reconhecem e investem para o desenvolvimento de ferramentas para abstração de conceitos técnicos ao programar para GPUs (ENGEL *et al.*, 2015).

Para haver um ganho de desempenho ao utilizar GPUs é necessário que a aplicação tenha um alto potencial de paralelização, pois os núcleos das GPUs trabalham em *clocks* mais baixos que os *clock* das CPUs. Muitas aplicações são extensas e foram validadas, o desenvolvimento de versões para GPUs destas aplicações exigiriam todo um redesenvolvimento delas e novas validações. A arquitetura de computadores segue um padrão que precisa de uma CPU gerenciando processos, mesmo em programação para GPUs a CPU é necessária organizando a comunicação entre as memórias, alocamento de memória, copias de dados. Criando assim um gargalo que pode ser impactante em aplicações pequenas. Ao desenvolver para uma GPUs a aplicação torna-se voltada para um *hardware* especifico limitando, as plataformas que a aplicação possa ser executada.

Na seção a seguir é apresentado um breve histórico da programação para propósitos gerais utilizando GPUs (GPGPUs).

2.4 PROGRAMAÇÃO COM GPUS

A ideia de utilizar GPUs para processamento pode parecer nova, mas foi concebida no início dos anos 90, sendo inicialmente limitada a funções matemáticas que passavam por pesadas adaptações para serem processadas nas GPUs, restringindo-se praticamente a rasterização e Z-buffers para acelerar tarefas como estimativa de custo em rotas e desenhos dos diagramas de Voronoi (SANDERS; KANDROT, 2004).

No início dos anos 2000, começou a produção de GPUs com unidades programáveis aritméticas chamadas *pixels shaders* que são responsáveis por funções processadas na GPU após a renderização da imagem, por exemplo efeitos como o de vidro, reflexo d'água, etc. Os parâmetros dessas funções geralmente são a posição da tela onde o pixel será exibido, e o que irá influenciar a aparência do pixel ao fim. Como os dados de entrada são controlados pelo desenvolvedor, pesquisadores começaram a utilizar estas unidades programáveis, com dados que ao invés de serem especificações para aparência final do pixel. Tais dados eram genéricos, como vetores e estruturas de dados. Em suma, as GPUs começaram a ser utilizadas para processar dados que não representassem gráficos, aproveitando o desempenho da unidades aritméticas que era superior ao encontrado nas CPUs, sendo este o advento da técnica conhecida como GPGPU(*General Purpose Graphics Processing Units*) (SANDERS; KANDROT, 2004).

No entanto, para conseguir programar utilizando a GPU, ainda era necessário que o pesquisador utiliza-se uma das APIs - Direct3D ou OpenGL. Isso era um problema, pois os pesquisadores nem sempre estavam familiarizados com a programação gráfica. Tal adaptação de *software* era necessária para conseguir converter o programa que era normalmente executado na CPU para ser executado na GPU. Por exemplo, era necessário modelar o problema quando envolvia fluxos de dados, para ser trabalhado como se fosse uma textura. O *kernel*, a função que seria aplicada independentemente a cada elemento do fluxo de dados deveria ser tratado como um *shader* [4]. A leitura do resultado do kernel aplicado a cada elemento do fluxo de dados deveria ser trado como a leitura da renderização de uma textura. O acesso a posição no fluxo de dados era tratado como a leitura de um pixel utilizando coordenadas, fazendo com que a programação com GPGPUainda fosse extremamente difícil de ser realizada.

Neste mesmo período, mais especificamente em 2003, durante o SIGGRAPH (*Special Interest Group on GRAPHics and Interactive Techniques*) foi apresentado o

BrookGPU, uma ferramenta para usar a linguagem de programação Brook, uma linguagem desenvolvida pela universidade de Stanford, como um compilador e interpretador em tempo real de fluxo de dados focando o processamento nas, então, modernas GPUs encontradas em placas gráficas da ATI e Nvidia.

O BrookGPUpermite compilar programas usando a sua linguagem de programação própria, a qual é uma variante de C ANSI. Podendo ser usado junto do Directx, OpenGL e AMD's Close to Metal como uma camada de *software*, executando tanto em Linux quanto em Windows. Há um simulador de placa de vídeo para *debugs* de códigos com esta ferramenta. O BrookGPU esteve em Beta por um longo período de tempo, não tendo mais atualizações desde 2007.

Embora tenha sido um grande avanço, existiam ainda muitas limitações. Era impossível, por exemplo, gerenciar a memória de uma forma eficiente, ou ter certeza da exatidão de dados com precisão dupla. Notando o sucesso que o BrookGPUalcançou a Nvidia contratou muitos dos pesquisadores envolvidos no projeto BrookGPU. A Nvidia como produtora da tecnologia das placas de vídeo GTX, detinha todo o conhecimento de como o Hardware era projetado e como ele estava funcionando na pratica, conhecendo a fundo o funcionamento das placas gráficas permitiu que os pesquisadores pudessem projetar uma tecnologia de programação para GPUs muito mais robusta e aproveitando melhor os recursos de Hardware presentes nas placas gráficas.

Em 8 de novembro de 2006 a NVIDIA lançou a GeForce 8800 GTX, que foi a primeira GPU criada com arquitetura CUDA(*Compute UnifiedCUDA Architecture*). Esta arquitetura incluiu componentes desenvolvidos especialmente para acabar com as limitações existentes, sendo ela uma arquitetura onde é possível o desenvolvimento de aplicações de propósito geral utilizando GPU. As ALU se tornaram programáveis e foram desenvolvidas segundo especificações da IEEE de precisão simples e dupla, terminado assim com os problemas da impressão de dados de ponto flutuante (KIRK; HWU, 2010).

Na seção a seguir é apresentado a Lei de Amdhal, um estudo teórico sobre a escalabilidade de projetos da computação paralela.

2.5 LEI DE AMDHAL E MEDIÇÃO DE DESEMPENHO

A Lei de Amdhal, ou Argumento de Amdhal, foi um trabalho teórico feito pelo famoso arquiteto de computadores Gene M. Amdhal. Apesar de erroneamente a Lei de Amdhal ser apresentada como um sinônimo para a formula do *speed up*, ela não aparece no trabalho original de Amdhal o qual ele apresentou para *AFIPS Spring Joint Computer Conference* em 1967 (AMDAHL, 1967).

Nesse trabalho, ele descreve características da computação em paralelo de maneira crítica a arquiteturas paralelas, expondo o que na época eram fatores limitantes para a paralelização de processos. No quarto paragrafo de seu trabalho, Amdhal cita os processos conhecidos como *housekeeping*, e como eles podem atrapalhar drasticamente o desempenho de uma aplicação paralela, por serem processos puramente sequenciais. Ao fim do já mencionado parágrafo Amdhal diz: "(...) uma conclusão bastante óbvia que pode ser tirada neste ponto é que o esforço gasto para alcançar altas taxas de processamento paralelo é desperdiçado, a menos que seja acompanhado por taxas de processamento sequenciais de quase a mesma magnitude(...) (AMDAHL, 1967)"

Esta afirmação é base para o argumento de que o *speed up* máximo de uma aplicação é limitado pela fração sequencial da mesma. O *speed up* é uma medida que mensura a performance de dois sistemas resolvendo uma mesma tarefa, apesar de ser uma medida que pode ser usada para medir qualquer alteração nos sistema, ela é amplamente utilizada para medir o ganho na obtenção do tempo de resposta de soluções paralelas. O *speed up* em p processadores é dado pela equação 1 (WITTEN; FRANK, 2015).

$$S(p) = \frac{Tseq}{Tpar} \tag{1}$$

Onde, Tseq é o tempo sequencial de execução do programa, e Tpar é o tempo de execução da versão paralela e p é o número de*threads* usados na solução paralela. Normalmente o *speed up* máximo possível de alcançar é de p*threads* para p núcleos, também conhecido como *speed up* linear (WITTEN; FRANK, 2015).

Um *speed up* acima disto é chamado *speed up* superlinear, somente ocorre em dois casos, um algoritmo sequencial sub ótimo, ou alguma outra alteração na configuração do sistema utilizado (mais memória disponível, processador mais potente, etc.).

A eficiência da paralelização é estimada medindo o tempo que os núcleos estão sendo efetivamente utilizados durante a computação. Sendo calculada pela equação 2 (WITTEN; FRANK, 2015).

$$e = \frac{Tseq}{Tpar * p} \tag{2}$$

Podemos ver que é possível substituir a equação 1 em 2.

$$e = \frac{S(p)}{p} \tag{3}$$

Ao observar a equação da eficiência, notamos que quando ocorre um *speed up* linear a eficiência obterá valor 1, equivalendo uma eficiência de 100%.

Na seção a seguir, serão apresentados os *profilers*, ferramentas capazes de identificar pontos de gargalo em *software*s.

2.6 PROFILERS

Com o surgimento da arquitetura distribuída ou dispositivos para diminuição no tempo de resposta como as GPUs modernas, é comum a intenção de reescrever todo o código para obter as vantagens oferecidas por esta arquitetura. No entanto, seguindo a Lei de Amdhal, nem todo algoritmo irá obter vantagem com uma versão paralela. Por este motivo um porte completo de uma aplicação já estabilizada, pode ser tanto custoso para seu desenvolvimento e ineficiente. Sendo uma grande verdade com a utilização de GPUs, uma vez que o custo da transferência de dados entre *threads* impacta no desempenho final da aplicação (ENGEL *et al.*, 2015), alguns exemplosde profilers são: Jprobe, Jprofiler, e a própria JVM.

Ao invés do trabalho massivo de reescrever todo o código de alguma grande aplicação para uma arquitetura *manycore*, é recomendado a utilização de ferramentas chamadas *profilers* para identificar os *hotspot* (pontos de gargalo) da aplicação, reduzindo assim o número de operação candidatas para serem reescritas (ENGEL *et al.*, 2015).

Profilers são ferramentas que permitem recolher informações da execução de aplicações, uma das utilidades dos *profilers* é avaliar diferentes elementos de um *software* e detectar quais trechos do código são mais computacionalmente onerosos (ENGEL *et al.*, 2015).

2.7 MINERAÇÃO DE DADOS

A mineração de dados (MD), surgiu da necessidade de agilizar o processo de descoberta de conhecimento em bases de dados, que com o passar do tempo ficam cada vez mais volumosas. Para entregar um resultado com baixo erro em um tempo aceitável (FAYYAD; PLATETSKY-SAPHIRO; SMYTH, 1996).

A MD é a etapa onde o conhecimento é descoberto na base de dados (KDD), em que algoritmos inteligentes são aplicados para extrair padrões que auxiliem a descoberta de conhecimento (HAN; PEI; KAMBER, 2011). O KDD é processo sequencial composto, geralmente por cinco etapas, como descrito a seguir (VELOSO, 2015):

- Seleção - Etapa onde é feita a compreensão do domínio dos dados e dos objetivos, da tarefa a ser realizada e a seleção de dados para formar o conjunto de dados com os atributos e exemplos a serem usados na tarefa;

- Pré-processamento - Etapa que consiste na limpeza dos dados, removendo ruídos e dados atípicos, tratamento de falta de dados e reconfiguração de dados a fim de garantir consistencia na base de dados;
- Transformação - Etapa onde é realizada a transformação dos dados em formatos utilizaveis (redução da dimensionalidade e discretização dos dados, etc);
- Mineração de Dados - Etapa onde é feita a escolha da tarefa de MD (Classificação, regressão, agrupamento, etc.) e é também nesta etapa que é feita a escolha de qual algoritmo de MD será aplicado na BD para realizar a extração de padrões, que podem ser apresentados de uma forma particular como, regras de associação, árvores de decisão, modelos de regressão, entre outras;
- Interpretação e avaliação - Por fim, é feita a interpretação dos padrões extraídos e a consolidação do conhecimento e comunicação aos interessados;

Toda e qualquer fonte de dados que apresenta padrões é candidata à MD (ex. Base de dados, históricos de medições, Web, dispositivos que coletam dados). Os padrões descobertos pela MD podem auxiliar na tomada de decisão e melhorar a acurácia de predições futuras (WITTEN; FRANK, 2015). A MD pode ser categorizada com base em quais tarefas serão realizadas, sendo elas descritas a seguir (WITTEN *et al.*,2016):

- Classificação (Classification) - Essa tarefa visa identificar a qual classe um determinado dado pertence. São algoritmos supervisionados porque analisam todos os registros de um conjunto de entrada os quais contem a indicação de qual classe eles pertencem, e aprendem como classificar um novo registro;
- Associação (Association) - A tarefa de associação visa encontrar relações entre os conjuntos de dados. Essa técnica de aprendizado é não supervisionada pois encontra qualquer regra de dados e não apenas as que predizem uma classe em especial;
- Agrupamentos (Clustering) - Tal qual a classificação o agrupamento é uma técnica de aprendizado não supervisionado que pode ser usada quando não há classes pré-definidas. Seu objetivo é identificar conjuntos presentes em um grupo de dados;

O agrupamento de dados é uma ferramenta difundida em diversos campos da ciência. Pode-se destacar entre suas aplicações na agricultura a segmentação de imagens (VIEIRA *et al.*, 2012), identificação de grupos de comportamentos dos ecossistemas (MILLS *et al.*, 2011) e delineamento de zonas de manejo agrícola (TAGARAKIS *et al.*, 2013). O K-meansé uma das técnicas mais populares de agrupamentos por ser eficaz e simples de ser implementada.

2.8 K-MEANS

O K-means é uma técnica que reúne n amostras em k grupos, de modo que as amostras dentro de um grupo sejam similares entre si e diferentes daquelas em outros grupos (SAKR.; GABER, 2014). A distância entre elementos do mesmo grupo é chamada de distância intragrupo e deve ser a mais baixa possível e a distância entre os elementos de grupos distintos é chamada de distância intergrupo e deve ser a mais alta possível (VELOSO, 2015). O Algoritmo K-meanssequencial é descrito resumidamente a seguir.

1. Seleção de K amostras como centroides iniciais.

2. Atribuição de cada amostra ao grupo com o centroide mais próximo.

3. cálculo de novos centroides

4. Passos 2 e 3 são repetidos até que se atinja o ponto de convergência.

Para estimar o quão parecidos são os elementos entre si são utilizadas técnicas para medir a distância entre estes elementos. A distância euclidiana é a medida mais utilizada. Em um espaço bidimensional, a formula da distância euclidiana entre dois pontos no espaço, podendo ser obtida pelo teorema de Pitágoras que fornece o segmento da reta que une dois pontos (BORTOLOSI, 2002):

$$d(a,b)(x,y) = \sqrt{(a-x)^2 + (b-y)^2} \quad (4)$$

Para um espaço n-dimensional, a distância euclidiana entre dois pontos p = p (p1, p2, p3, ...,pn) e x = (x1,x2,x,3,. ,xn) é calculada por (BORTOLOSI, 2002):

$$d(x,p) = \sqrt{(x1-p1)^2 + \cdots + (xn-pn)^2} = \sqrt{(xn-pn)^2}\ \omega\omega\ (5)$$

O tempo necessário para executar o K-means é diretamente proporcional ao tamanho da base de dados n, ao número de grupos k e a dimensionalidade da base de dados. São necessários k x n cálculos em cada iteração (BEKKERMAN; BILENKO; LANGFORD, 2012). O K-means tem perfil escalável, quando um algoritmo tem a característica de usar mais de um núcleo para resolver o problema de maneira eficiente. As amostras podem ser divididas em estruturas de dados que serão distribuídas para cada um dos núcleos e então atribuídos ao grupo com o centroide mais próximo, em paralelo. (DEAN, 2014). O K-meansapresenta algumas limitações:

- Os grupos finais são influenciados pelos centroides iniciais, os quais normalmente são escolhidos de maneira aleatória. Mesmo pequenas alterações nos centroides escolhidos no início podem prover grupos diferentes. Ademais, é difícil estimar o número correto de grupos em um domínio desconhecido (VELOSO, 2015);
- Pode ser necessário alguma normalização antes que os dados sejam agrupados. Por exemplo, conversão de unidades de alguns atributos equivalentes (VELOSO, 2015);

Uma forma para estimar o número de grupos em uma base de dados desconhecida é estimar a distância entre os elementos dos grupos resultantes. De maneira que seus elementos sejam os mais distantes dos elementos de outros grupos e mais próximos possíveis entre si (DUNN, 1974).

Podem ser usados alguns algoritmos de agrupamento como o *Canpopy* para selecionar um grupo inicial de centroides (MCCALLUM; NIGAM; UNGAR, 2000). Normalmente, algoritmos como o K-meanspodem ser usados através de ferramentas de mineração de dados, como a ferramenta (WITTEN *et al.*, 2016). Esta ferramenta é descrita com maiores detalhes na próxima seção.

O K-means é um algoritmo utilizado em estudos com MD por sua fácil implementação, por ser um problema NP-Difícil ele muitas vezes é resolvido através de heurísticas avançadas, mas que em muitos casos ainda é computacionalmente in- tenso mostrando assim um alto potencial para trabalhar com computação paralela para reduzir seu tempo de resposta.

2.9 WEKA

WEKA (*Waikato Environment for Knowledge Analysis*) é um pacote de *software* escrito em Java, que tem por objetivo agregar diversos algoritmos de inteligência artificial, como por exemplo algoritmos de *machine learning e Datamining*, a ferramenta teve seu desenvolvimento iniciado em 1993 na Universidade de Wakaito na Nova Zelândia. Sendo ele um *software* livre, licenciado sobre uma licença GNU (*General Public License*). Apresentando também uma coleção de ferramentas para a visualização dos dados obtidos pelo Weka (WITTEN *et al.*, 2016).

A versão original do Weka era um front-end de Tcl/Tk para algoritmos de modelagem (principalmente de terceiros) implementados em outras linguagens de programação, além de utilitários de pré-processamento de dados escrito em C e um sistema baseado em Makefile para executar experimentos de aprendizado de máquina. Esta versão foi projetada como uma ferramenta para analisar dados de domínios agrícolas (HOLMES; DONKIN; WITTEN, 1994) em linguagem C. A versão mais recente é total-

mente escrita em Java (Weka 3), cujo desenvolvimento começou em 1997 (HOLMES; DONKIN; WITTEN, 1994).

O Weka suporta várias rotinas de mineração de dados, mais especificamente: pré-processamento de dados, *clustering*(agrupamento), classificação, regressão, visualização e seleção de recursos. Todas as rotinas do Weka baseiam-se no pressuposto de que os dados estão disponíveis como um arquivo simples ou relação, onde cada ponto de dados é descrito por um número fixo de atributos (normalmente, atributos numéricos ou nominais, mas outros tipos de atributos também são suportados). O Weka também fornece acesso a bancos de dados SQL usando o *Java Database Connectivity* e pode processar o resultado retornado por uma consulta no banco de dados. O Weka fornece acesso a *Deeplearning* com o Deeplearning4j[6].

O código fonte do Wekaé disponibilizado para download para que desenvolvedores possam baixá-lo e modificá-lo. Para compilar o código fonte do Wekaé necessário utilizar o Ant, uma ferramenta desenvolvida pela Apache, a qual será abordada na seção a seguir.

2.10 ANT

O Ant (*Another Neat Tool*) é uma ferramenta utilizada para a automação de compilação e construção de *softwares*, que originou-se do Apache Tomcat. Quando o desenvolvedor do Tomcat, James Duncan Davidson, estava desenvolvendo o Tomcat ele notou que o make, utilitário para compilação de códigos, apresentava algumas inconsistências dependo de em qual ambiente o usuário iria construir o Tomcat, para resolver este problema Duncan desenvolveu, em Java, uma ferramenta para construção do Tomcat em qualquer ambiente, esta ferramenta é o Ant [7].

Por ser uma ferramenta feita em Javaele é executado pela *Java Virtual Machine* tornando-a uma ferramenta que independe de plataforma. Além disso, é uma ferramenta que possui melhor desempenho ao construir outras aplicações Java. Enquanto o Make, executa uma série de comandos do Sistema Operacional, o que dificulta a sua portabilidade, o Ant resolve isto utilizando uma série de funções do próprio Ant, garantindo assim que as funcionalidades irão comportar-se de maneira idêntica em qualquer ambiente(BURKE; TILLY, 2002).

O Ant utiliza um xml, que por padrão deve ser nomeado como "build.xml", ele irá conter uma série de tarefas para a construção da aplicação as quais serão chamadas com as *tags*. No Ant, o xml é usado como uma linguagem de programação ao invés de

simplesmente conter dados brutos, isto gera algumas críticas por parte da comunidade sobre a arquitetura do Ant[8].

No apêndice deste trabalho encontra-se como configurar corretamente o Ant para construir o Weka tanto no linux quando no windows, em ambos os casos os comandos para a construção são os mesmos, mudando apenas a instalação e a configuração da ferramenta. Na seção a seguir é apresentado o CUDAferramenta para a programação GPGPU.

2.11 CUDA

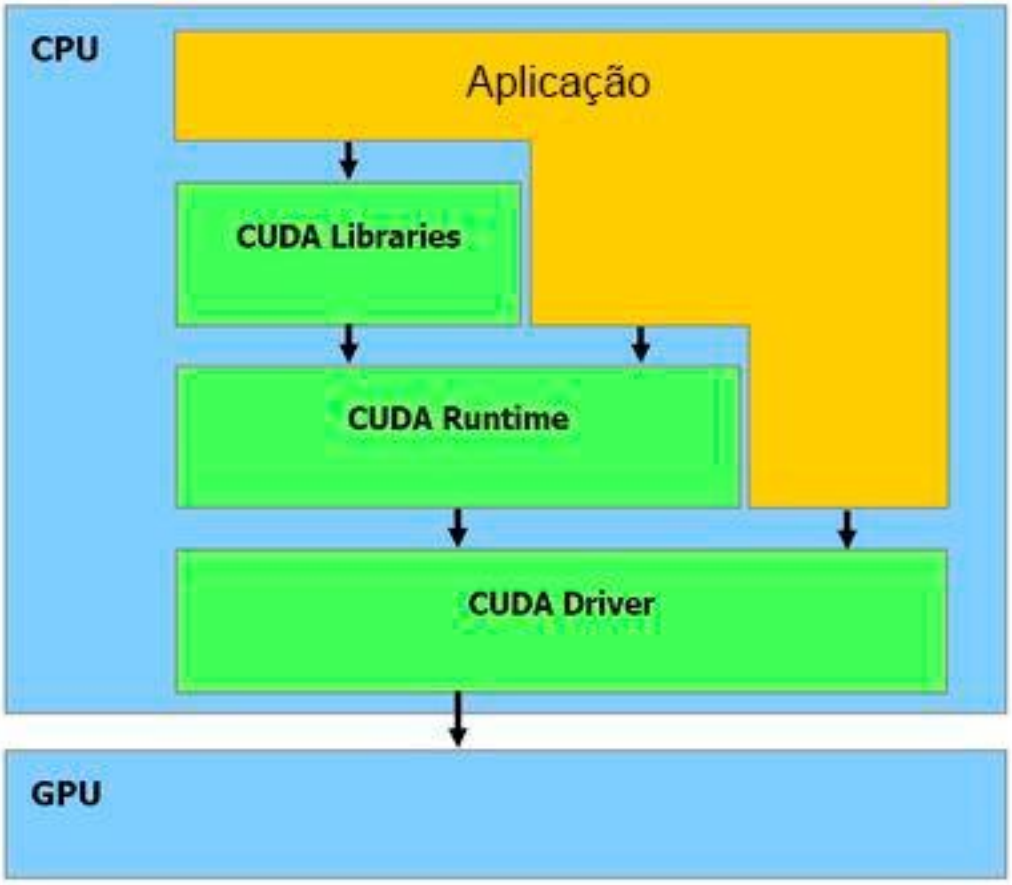
CUDA (anteriormente conhecido como *Compute UnifiedCUDA Architecture*), é uma plataforma para computação paralela e uma API(*application programming interface*) destinada para o controle da GPU. CUDApermite que o desenvolvedor possa acessar a GPU e utilizar ela para propósitos gerais e não somente relacionados à processamento gráfico, por exemplo com CUDAé possível trabalhar com inteligência artificial ou reco- nhecimento de padrões, tarefas as quais a GPU originalmente não foi concebida para realizar, dessa forma tornando ainda mais presente o termo GPGPU, que é a GPU sendo usado para diferentes fins e não apenas para processamento gráfico. O CUDAfunciona como uma camada de *software*s responsável por permitir a execução de kernels de código em ambiente paralelo[9].

A API, inclui um conjunto de instruções nativas ao próprio CUDA e diversas bibliotecas com funções de apoio para o desenvolvedor. Ao programar-se em CUDA fica delegado ao desenvolvedor configurar os acessos a memória global, a cache, a quantidade e a disposição dos*threads*. O desenvolvedor também será responsável por escalonar as atividades entre a GPU e o CPU. CUDAfoi projetado para ser trabalhado nativamente com as linguagens C, C++, Fortran. Além destas linguagens existem atualmente projetos para desenvolver em CUDA utilizando as seguintes linguagens: Common Lisp, Clojure, F#, Haskell, IDL, Java, Julia, Lua, Mathematica, MATLAB, .NET, Perl, Python, Ruby, R.

Inicialmente o CUDAera utilizado apenas para cálculos de física em jogos, identificar ateromas e estimativa de trafico aéreo[10]. Atualmente CUDAalém destas aplicações, é utilizado em análise de risco para desastres naturais, simulações climáticas, investigação de fraudes financeiras[11].

O *software* CUDApossui três camadas de *software*, como pode ser observado na figura 3:

Figura 3 – Nvidia CUDAArquitetura de *software*



Fonte: CUDA *software Stack* (NVIDIA, 2007)

CUDAfornece duas APIs:

- Uma API de alto nivel: A CUDARuntime API.
- Uma API de baixo nivel: A CUDADriver.

Como a API de alto nível é implementada “acima” da API de baixo nível, cada chamada para uma função do Runtime é dividida em instruções mais básicas gerenciadas pela API do driver. As APIs são mutuamente exclusivas, ou seja, o desenvolvedor escolhe ou uma ou outra. Porém, nunca ambas, sendo impossível mesclar funções de ambas as APIs. Mesmo a Nvidia tendo escolhido chamar a Runtime API de "API de alto nível"ela apresenta muito elementos que podem ser considerados de baixo nível; apesar de oferecer muitas funções de alto nível para o gerenciamento de contexto da memória.

A Driver API também fornece uma comunicação com DirectX, principalmente porque em funções de geração de formas geométricas, cálculos físicos, fluidos ou geração de elementos automáticos dos mapas de jogos podem ser feito em tempo real utilizando a estrutura fornecido pelo CUDAEsse mesmo sistema pode também ser usado em visualização de simulação de fluídos, simulação de fenômenos climáticos, etc. Também pode-se utilizar este sistema para trabalhar com programação de baixo nível, como era feito antes mesmo da disponibilidade da BrookGPU - utilizando Directx e OpenGL- isto pode ser útil quando quer evitar o gargalo do PCI Express. Porque os dados ficam todos armazenados na VRAMda placa gráfica ao invés de ser armazenado na memória RAM.

A API de programação CUDAapresenta uma série de peculiaridades e definições de sua arquitetura que devem ser levados em consideração para o entendimento da estrutura de códigos em CUDApeculiaridades e definições que serão abordadas a seguir. Na seção a seguir são apresentados alguns conceitos e nomenclaturas do CUDA

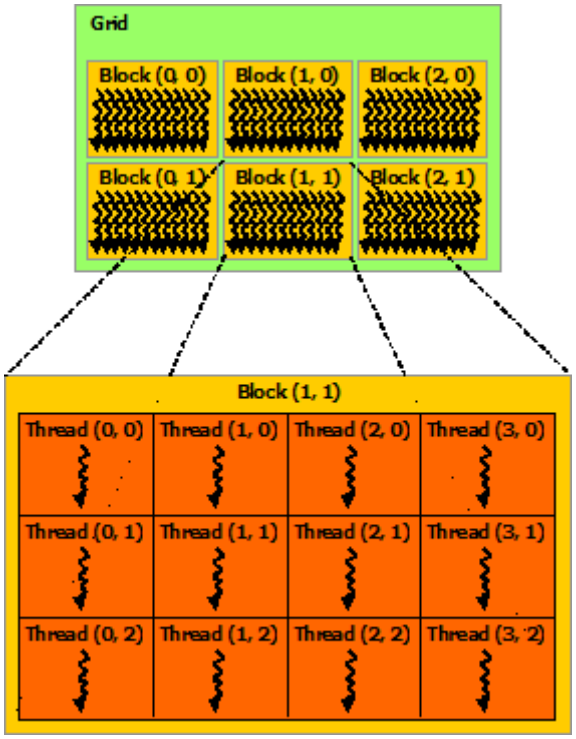
2.11.1 Host, CUDA, *threads*, Blocos e Grids

Device, é a placa gráfica como um todo incluindo a GPU e a memória VRAM da placa. **Host** é como é referenciado na documentação do CUDAa CPU e a memória RAM, a comunicação entre*host* eCUDAno CUDAé feita através do PCI-express.

Em CUDA*threads* são diferentes das*threads* da computação paralela utilizando apenas a CPU. Em CUDA elas são um elemento básico de dados para serem processados. Diferente das*threads* de CPU elas são extremamente leves, fazendo com que uma mudança entre duas*threads* não tenha um custo operacional alto. Tamanha granularidade pode torna difícil para o desenvolvedor trabalhar, para isto as*threads* são agrupadas em blocos de*threads*, que atualmente podem conter até 1024*threads* por bloco. Este número é limitado, pois o CUDAprecisa garantir que todas as*threads* de um bloco sejam executadas no mesmo para ter acesso a memória compartilhada entre si. Os blocos podem ser organizados em matrizes de até três dimensões (x,y,z).

Grids são conglomerados de blocos que podem ser organizados como uma matriz de duas dimensoes (x,y). A figura a seguir mostra um grid de duas dimensoes(x=3 e y=2) sendo formado por 6 blocos de duas dimensoes(x=4 e y=3). Cada bloco possui 12*threads*.

Figura 4 – Esquema de Grids, blocos e*threads*



Fonte: CUDA Programing Guide

Na subseção a seguir é apresentado o *kernel*, o método que será executado em paralelo no *device*.

2.11.2 kernel

kernel é como é chamado o método inicial que será executado noCUDA, cada vez que o *host* precisa chamar uma função ou método da GPU, o kernel é chamado. O kernel é onde pode-se chamar funções do próprio CUDA São necessários utilizar identificadores para que o compilador CUDAdiferencie funções que serão executadas no *host* e quais no CUDA. O identificador **_global_**, sinaliza que a função será chamada pelo *host* e executada pelo CUDA, iniciam com o identificador**_device_**as funções que serão chamadas e executados noCUDA(NETO, 2014). Na seção a seguir é apresentado como é feito a declaração de um *kernel*

2.11.3 Declaração de um kernel

O*host* sempre é o responsável pela chamada de um kernel, utilizando alguma das linguagens suportadas pelo CUDA Ao fazer na chamada de um kernel deve-se sempre definir quantos blocos e*threads* serão executados noCUDA(NETO, 2014).

A seguir a definição de um kernel na linguagem C (SANDERS; KANDROT, 2004)
.

Codigo fonte 1: Chamada de um kernel genérico.

```
nome_kernel <<num_blocos , num_threads >>(param1 , param2 , . . . ) ;
```

Nome_kernel, é o nome da função que será executada pela GPU, num_blocos é o número de blocos que serão usados e num_threads é o número de*threads* por bloco, os "params"são os parâmetros das funções que devem ser declarados nas próprias funções. A definição do kernel deve contar com o identificador_global_. O código fonte 2 mostra a definição de um kernel genérico (SANDERS; KANDROT,2004).

Codigo fonte 2: Definição de um kernel genérico.

```
__global__ nome_kernel ( param1 , param2 , . . . ){
.
.
.
}
```

Normalmente os parâmetros recebidos no kernel são ponteiros ou dados. A seguir no código fonte 3, temos um programa simples em CUDApara somar dois vetores:

Codigo fonte 3: Soma de Vetores em C utilizando CUDA

```
// kernel definition
 __global__ void VecAdd(float* A, float* B, float* C){i
 nt i=thread Idx .x;
C[i]=A[i]+B[i];
 }

int main(){
...
// kernel invocation with N\textit{threads
}VecAdd<<<1 , N>>>(A, B, C);
...
}
```

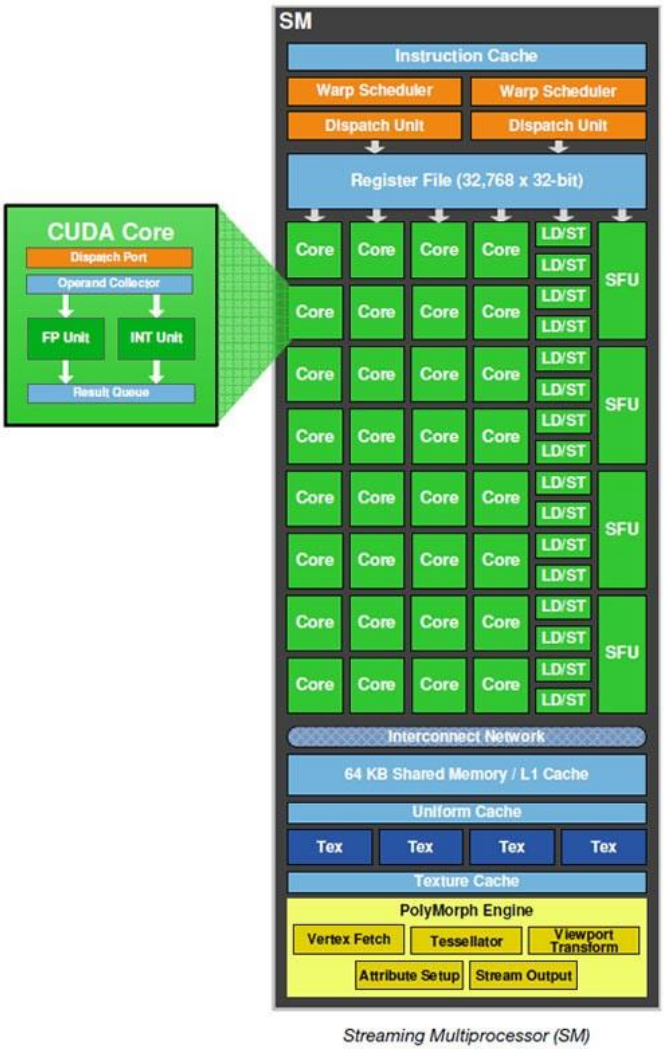
Na seção a seguir é apresentado a estrutura de um *Stream-Multiprocessor*.

2.11.4 SM e SP

GPUs implementam unidades chamadas SM (*Stream-multiprocessor*). Cada uma destas unidades implementa muitos SP(*stream-processors*) que são conhecidas

como núcleos CUDAou *cuda-cores*. Para que os SM sejam ativados é necessário que recebam um bloco de*threads*. Na figura a seguir podem ser vistos os componentes de um SM.

Figura 5 – Arquitetura do SM



Streaming Multiprocessor (SM)

Fonte: CUDA Programing Guide

Os seus principais componentes são (NETO, 2014):

Warp Scheduler: Responsável pelo escalonamento das*threads* nos núcleos CUDA

Register File: É a memória dos registradores e é a memória mais rápida dentro de uma GPU.

Core: São os núcleos CUDAou CUDAcores. Os núcleos são responsáveis pelo processamento das*threads*. Dentro de cada núcleo existem unidades responsáveis pelo processamento de inteiros e ponto flutuante.

LD/ST: É a abreviação de load/store. Sempre que é necessário acesso a memória da GPU, essas unidades são usadas.

SFU: É um acronomo para *Special Function Unit*. É a unidade responsável por realizar operações matemáticas, como secante, cossecante, exponencial, etc.

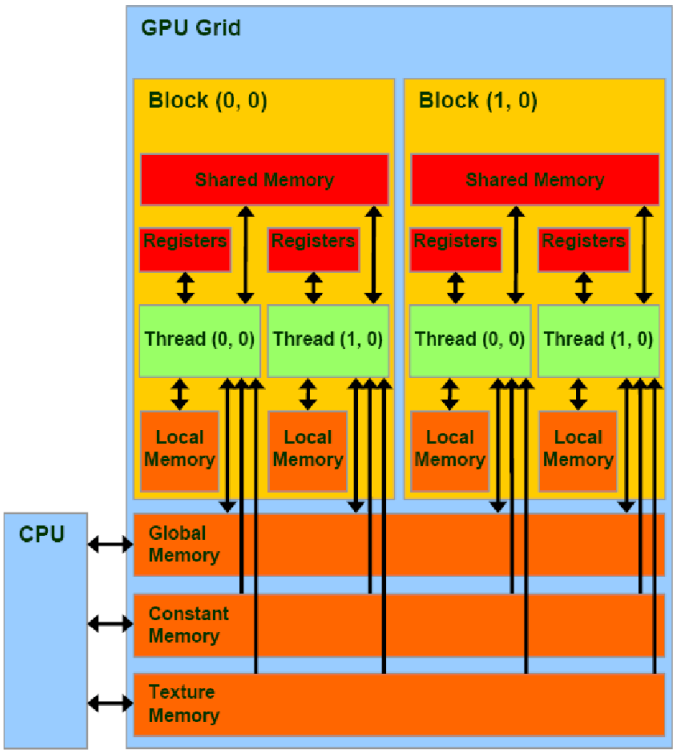
Shared Memory: É a memória compartilhada dentro de um bloco. Todos os *cores* de um SM tem acesso a essa memória.

A quantia de SMvaria de acordo com o modelo das placas gráficas. O modelo 1080TI possui atualmente 28 SM. Na subseção a seguir é apresentado a organização da arquitetura de memória CUDAseus componentes e suas funções.

2.11.5 Arquitetura de memória CUDA

Em CUDAexistem 6 diferentes categorias de memórias, variando desde as mais rápidas e mais escassas até as mais abundantes e lentas. É primordial ao desenvolver uma aplicação para GPGPU , que o desenvolvedor saiba os tipos de memória a sua disposição e qual a função de cada uma destas memórias (NETO, 2014), uma vez que em CUDA é função do programador fazer o gerenciamento das memórias. A próxima figura mostra os tipos de memória disponíveis ao trabalhar com CUDA

Figura 6 – memórias em CUDA



Fonte: CUDA Programing Guide

Como pode ser observado na figura 6 a Arquitetura de memória é dividia em

(NETO, 2014):

Registers: O registrador é a memória mais rápida da GPU. É um tipo de memória que se limita a thread que está executando, sendo assim uma*threads* não tem acesso, seja de escrita ou leitura, em um registrador de outra *thread*, ele está localizado na própria GPU e não é acessível ao*host*.

***Shared Memory* :** É uma memória acessível de qualquer thread de um bloco, cada SM possui um banco de cada uma destas memórias. O tempo de acesso é baixo, uma vez que a memória compartilhada também se encontra dentro da GPU. O *host* não possui acesso direto à memória compartilhada.

***Local Memory* :** Assim como os registradores, o escopo da memória local é apenas da *threads*. A diferença é que a memória local fica fora da GPU, o que acarreta em um tempo de acesso alto. Essa memória, foi projetada para armazenar grandes estruturas de dados, como arrays e matrizes muito longas. O*host* possui acesso indireto a esta memória.

***Global Memory* :** É a memória global para a GPU trabalhar, *threads* de diferen- tes blocos tem acesso a ela. por se tratar de uma memória que também está localizada fora da GPU, o que acarreta em um tempo grande para o acesso. Tanto o*host* quando oCUDAtem acesso a essa memória.

***Constant Memory* :** É uma memória utilizada para armazenar valores que serão constantes durante toda a execução do kernel, tendo um tamanho de aproximadamente 64 KB, podendo ser usada para armazenar constantes durante a execução do kernel. Apesar de ser uma memória localizada fora do chip da GPU, seu tempo de acesso é baixo por ser extremamente otimizado, apesar que se houver um erro ao acessar o endereço exato da constante é necessário percorrer a memória para encontrá- la, podendo ter de percorrer toda a memória constante para achá-la no pior caso, acarretando assim em uma brusca queda de desempenho. No trecho de código a seguir temos o exemplo de uma declaração para usar a memória constante.

Neste exemplo foi declarado uma *array* de números com ponto flutuante na memória constante. Tanto o *host* quanto o CUDAtem acesso a essa memória.

***Texture Memory* :** É uma memória otimizada para armazenar texturas 2D (Uma abstração de uma superfície em uma matriz com valores que numéricos (CATMULL, 1974)). As buscas dessas texturas são eficientes, pois existe memórias caches que armazenam os endereços. Essa memória pode ser usada em programação da GPGPU usando DirectX e OpenGLcomo era feito antes do lançamento do BrookGPU, isso pode ser proveitoso por utilizar o acesso extremamente rápido dessa memória. Tanto o *host* quanto o CUDAtem acesso a essa memória.

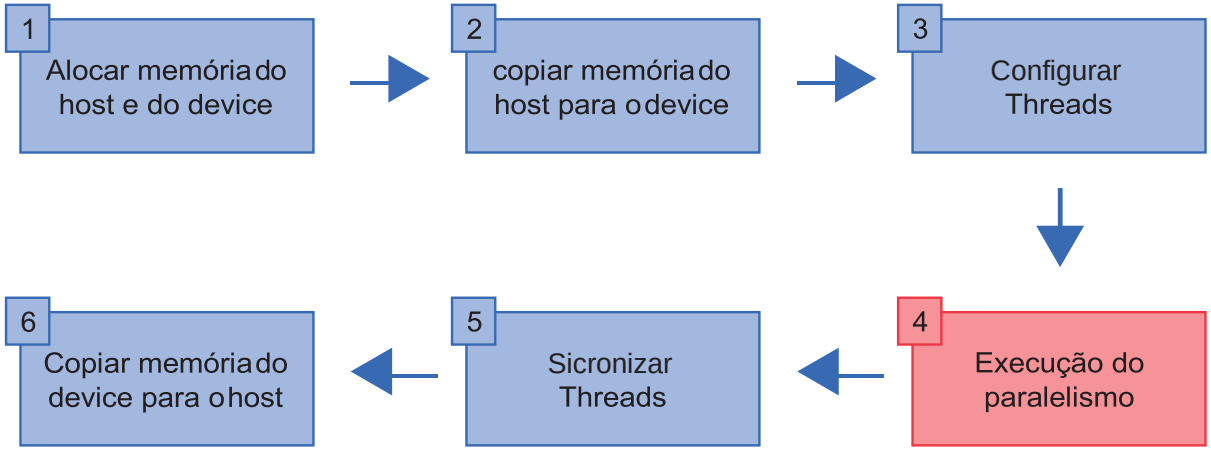
Na seção a seguir é exemplificado como é feito o fluxo de processamento

durante a execução de uma aplicação feita para rodar no CUDA

2.12 FLUXO DE PROCESSAMENTO

O *device* não precisa ter acesso a todo o código, precisa ter acesso apenas ao kernel, que será copiado do*host* para oCUDA. O fluxo de processamento de uma aplicação GPGPUé apresentado na imagem 7.

Figura 7 – Fluxo entre *host* e CUDA



Fonte: O autor

Ao observar o fluxograma anterior percebe-se que o primeiro passo é alocar a memória, tanto no *host* quanto no *device*. O espaço alocado deve ser o mesmo no *host* e *device*, pois os dados do *host* serão copiados para o *device*, como é visto no passo 2. No passo 3 é feita a definição de quantas *threads* e blocos serão utilizados durante a execução. Após a execução das *threads* é necessário fazer a sincronização das *threads* para que possa ser feita a cópia dos resultados do *device* para o *host*, para que finalmente possa dar continuidade na aplicação, ou então mostrar ao usuário o resultado (NETO, 2014).

O desenvolvimento em CUDA é potencialmente complicado, demandando um bom entendimento do algoritmo a ser paralelizado, uma reescrita de trechos complexos, alto grau de entendimento do funcionamento de placas gráficas e suas arquiteturas, entendimento das funcionalidades do CUDA e da arquitetura de memória do próprio CUDA. Juntamente com o fato do CUDA ser projetado pela NVIDIA para atuar naturalmente apenas nas linguagens C, C++ e Fortran, há uma necessidade de desenvolvimento de ferramentas para utilizar CUDA em outras linguagens de programação e para abstrair conceitos técnicos da programação.

Na seção a seguir, será abordado o Nvidia-SDK, um conjunto de ferramentas para desenvolvedores poderem estender o CUDA para outros compiladores e linguagens.

2.13 NVIDIA-SDK

A programação para CUDA é complexa, de acordo com a própria Nvidia, ela mesma oferece ferramentas para que seja possível fazer com que compiladores de novas ou já existentes linguagens de programação usem o CUDA inclusive para criar funções novas abstraindo conceitos técnicos da programação com CUDA

Isto é possível porque o nvcc, o compilador padrão da linguagem CUDA foi desenvolvido usando a LLVM, que é uma infraestrutura para geração de compiladores, escrita em C++ desenvolvida principalmente para realizar a compilação, ligação e execução de programas escritos em linguagens de programação variadas. É uma ferramenta originalmente projetada para gerar compiladores da linguagem C e C++. Porém, sua arquitetura tornou capaz que fosse expandido as linguagens suportadas e atualmente inclui as seguintes linguagens de programação: Objective-C, Fortran, Ada, Haskell, bytecode, Java, Python, Ruby, ActionScript, GLSL, Julia, entre outras. Foi originalmente desenvolvida pela equipe de pesquisa da Universidade Illinois, atualmente é mantido pela Apple, fazendo inclusive parte do kit de desenvolvimento do IOS[12].

O nvcc, ao ter sido desenvolvido com essa ferramenta permite que desenvolvedores possam estender funcionalidades do Nvidia-SDK para seus compiladores podendo assim ter suporte integrado a computação paralela. Além da própria estrutura do compilador para isto contam com bibliotecas que auxiliam neste desenvolvimento, os quais serão listados a seguir[13]:

- Um conjunto de bibliotecas, libdevice.*.bc, que implementam funções matemáticas recorrentes no formato LLVM bitcode;
- Um conjunto de exemplos que ilustram o uso do compilador SDK;
- A documentação para o Compilador SDK (incluindo a especificação para LLVMIR, a documentação da API libnvvm e a documentação da API para libdevice), pode ser encontrados no subdiretório "doc"ou *on-line*;
- As bibliotecas otimizadas do compilador, a biblioteca idevice pode ser encontrada no subdiretório nvvm, com exemplos de como utilizá-la;

- Uma biblioteca de compiladores otimizada (libnvvm.so, nvvm.dll / nvvm.lib; libnvvm.dylib) e seu arquivo de cabeçalho nvvm.hsão fornecidos para desenvolvedores de compiladores que desejam gerar PTX a partir de um programa escrito em NVVM IR, que é um compilador representação interna baseada em LLVM;

Mais códigos de exemplos de como utilizar o Nvidia-SDK podem ser achados no repositório da Nvidia no github[14].

Nas seção a seguir serão abordados algumas ferramentas, que utilizam o Nvidia-SDKpara implementar CUDApara outros compiladores, e ou, bibliotecas.

2.14 IBM-SDK E JIT

Ao decorrer dos últimos anos foram desenvolvidos muitos projetos para estender o CUDA para outros compiladores e bibliotecas. Podemos citar o JavaGPU, JCUDA, Aparari, RootBear, IBM-SDK. Todos são esforços para tentar tornar a programação com GPGPUmais acessível abstraindo conceitos técnicos. Para este trabalho a ferramenta escolhida foi o IBM-SDK.

O IBM-SDK, é o conjunto de ferramentas fornecidos pela IBM-SDKpara o desenvolvimento em Java, sendo ele uma alternativa ao JDK da oracle e ao Open JDK que é instalado automaticamente nas distribuições Linuxao instalar o pacote Java. O IBM-SDK atualmente,2018, tem suporte para compilar código em Java 8 apenas para plataforma Linux, para Windows o IBM-SDKainda compila códigos apenas para o Java 7.

A partir do Java 8, há uma nova estrutura de repetição, que é a possibilidade de poder utilizar Streams ao invés de usar as estruturas de repetição tradicionais. Junto da nova funcionalidade, foram implementados muitos métodos que podem ser invocados a partir desta nova estrutura poupando tempo para o desenvolvedor e encapsulando *exceptions* que podem ocorrer durante a execução. Isto trouxe muita liberdade para os desenvolvedores trabalharem, pois agora eles podem instruir o que deve ser feito com a estrutura de repetição e não precisando se preocupar com como isso deve ser feito. Há na estrutura *Stream*, suporte ao paralelismo, como pode ser visto exemplo a seguir:

Codigo fonte 5: StreamInt.

```
public static void main(String[] args){
  IntStream stream = IntStream.range(5, 12);
```

```
        System.out.println("The corresponding " +
                           "parallel IntStream is :");
        stream.parallel().forEach(System.out::println);
    }
```

No trecho de código anterior. Foi primeiramente definido o tamanho da *Stream*, o qual seria "7"(com o limite inferior sendo 5 e o superior 12), depois cada um dos elementos foi executado simultaneamente de maneira paralela.

Essa nova utilidade do Java permite com que desenvolvedores possam abstrair conceitos que anteriormente precisavam ser explicitamente programados, como a criação das*threads*, sua sincronização, etc.

Uma das ferramentas presentes no IBM-SDKé o JIT(*Just in Time Compiler*), é uma ferramenta para auxiliar um código executado com o JDK da IBM. É uma ferramenta que analisa em tempo real o desempenho do código que está executando e aplica-se heurísticas constantemente para avaliar o desempenho do código que está sendo executado, e poder alterar de maneira automática a execução do código da CPU para a GPU.

O funcionamento do JIT ocorre da seguinte forma, quando um método é escolhido para compilação, a JVM alimenta seus *bytecodes* para o JIT. O JIT precisa entender a semântica e a sintaxe dos *bytecodes* antes de poder compilar o método corretamente.

Para ajudar o compilador JITa analisar o método, seus *bytecodes* são primeiro reformulados em uma representação interna chamada "trees", que se assemelham mais ao código da máquina do que aos *bytecodes*. A análise e as otimizações são então realizadas nas árvores do método. No final, as árvores são traduzidas em código nativo.

O compilador JIT pode usar mais de um encadeamento de compilação para executar tarefas de compilação JIT. O uso de vários encadeamentos pode ajudar potencialmente os aplicativos Java a serem iniciados mais rapidamente. Na prática, vários encadeamentos de compilação JITmostram melhoras de desempenho somente quando há núcleos de processamento não utilizados no sistema.

O número padrão de encadeamentos de compilação é identificado pela JVM e depende da configuração do sistema.

Para que seja possível usar o JIT da IBM-SDK deve-se fazer uma serie de preparações as quais serão listadas e explicadas a seguir:

- **Escrita do Código:** Para que um trecho computacionalmente intenso possa ser automaticamente executado na GPU ao invés da CPU pelo JITele precisa ser

escrito usando a estrutura de repetição *Stream* e explicitar para ser executado de forma paralela, com os limites superior e inferior definidos de maneira explicita e constante, com um consumidor para cada iteração que deve ser expressa em um *lambda*, como pode ser visto no exemplo 5:

Codigo fonte 5: Stream paralela.

```
...
IntSTream.range(0, N).parallel().foreach(i -> {b
[i] = a[i] * 2;
})
...
```

No trecho de código anterior é definido os limites superior e inferior(0 e N), após isto foi explicitado para ser executado em paralelo criando N *threads*, é declarado o consumidor i, e após isto o que cada*threads* irá executar é definido no lambda da função (Que neste exemplo é definido como: b[i] = a[i] * 2).

- **Limitações do JIT:** o JITao ser usado para o paralelismo ainda possui diversas limitações podendo serem citadas como as principais: Necessidade de constantes na definição das *threads*. Não consegue realizar chamadas de métodos nas execuções das*threads*. Só consegue trabalhar com tipos primitivos do Java não podendo usar objetos, nem fazer chamadas de métodos, também não há acesso entre as *threads* (devido a limitações do próprio CUDA).

  Ele conta com uma longa e complexa documentação a qual deve ser estudada a fundo para conseguir utilizá-lo corretamente, por exemplo: Ele por padrão não utiliza GPU no processamento a não ser que seja explicitado na linha de comando, aplicações leves também costumam ativar a placa de vídeo precisando ser explicado que sejam executadas nas

3 IMPLEMENTAÇÃO DO ALGORITMO K-MEANS EM AMBIENTE *MANY-CORE*

No geral, o trabalho consistiu nas seguintes etapas:

1. Configuração do Ambiente de trabalho;
2. Escolha da base de dados;
3. Escolha do Profiler para ser usado.
4. Identseicação do método gargalo do K-means no Weka;
5. Reescrita do método para a versão paralela;
6. Realização de medição de desempenho.

O ambiente utilizado para o experimento foi um computador com sistema Operacional Ubuntu, processador i5 7400, 4 núcleos e 4 threads com clock de 3.5 Ghz em *Stock* e trabalhando a 4.0 Ghz em modo *Turbo*. Memórias Ram com 8Gb funcionado com frequência de 2.4 Ghz. Placa gráfica Nvidia GTX 1060 6GB, com 1280 cuda cores, clock dos CUDAcores de 1.708 Ghz e largura de banda de 196 Gb. A versão utilizada do CUDA foi a 10.0.130.

3.1 CONFIGURAÇÃO DO AMBIENTE DE TRABALHO

A configuração do ambiente de trabalho, foi uma parte muito trabalhosa devido a incompatibilidade de ferramentas utilizadas, precisando muitas vezes de bastante configuração manual das mesmas para poderem ser integradas uma a outra. No Apêndice A deste trabalho encontra-se em detalhes estas configurações.

3.2 ESCOLHA DA BASE DE DADOS

A base de dados escolhida para o trabalho foi uma base de dados disponível para MD contendo dados referentes a produtividade de soja em áreas agrícolas com 80 atributos dseerentes, 6 classes e 12800 instanciância. Essa base foi escolhida por ser a base com o maior tempo para obtenção de resposta que foi encontrada, podendo assim verseicar a capacidade da GPU em escalar com

grandes espaços amostrais. Além disso, esta base de dados tem Característica comumente encontradas nos trabalhos de mineração de dados e computação aplica à agricultura.

3.3 ESCOLHA DO PROFILER PARA SER USADO

Profilers são ferramentas que permitem recolher dados da execução de aplicações. Uma das utilidades dos *profilers* é avaliar dseerentes elementos de um software e detectar quais trechos do código são mais computacionalmente intenso (ENGEL *et al.*, 2015). Para este trabalho foram pré-selecionados três profilers: JProfiler, Jprobee a ferramenta integrada da JVM (*Java Virtual Machine*). As três ferramentas identseicaram o método moveCentroidscomo sendo o método gargalo. O dseerencial entre as ferramentas foi o tempo para conclusão da análise.

O *profiler* monitora os processos ativos na JVM, ao encontrar algum processo, o *profiler* apresenta o tempo gasto na atual linha do tempo de cada um dos métodos das classes que estão sendo executadas na JVM, ou seja, os *profilers* apresentam dados atualizados em tempo real. Os três *profilers* escolhidos indicaram um mesmo método como sendo o gargalo na execução do *K-means*, variando o tempo para a conclusão da execução do *K-means* quando utilizando os *profilers*.

Na tabela 1 há um comparativo entre os tempos de execução com cada um dos *profilers* pré-selecionados:

Tabela 1 – Comparativo dos tempos de execução dos *Profilers*

Execução	Sem profilers JVM 1	Jprofiler 00:02:26	Jprobe 00:08:06	
00:15:07	-			
2	00:02:22	00:08:00	00:15:28	-
3	00:02:20	00:08:12	00:15:06	-
4	00:02:25	00:08:12	00:15:20	-
5	00:02:30	00:08:04	00:15:14	-
Media	00:02:25	00:08:06	00:15:14	Inconclusivo
Desvio padrão	3,485	4,782	8,344	Inconclusivo

Fonte: O autor

Como pode ser observado na tabela 1 utilizar um *profilers* vai tornar a conclusão da tarefa mais demorada, isto acontece porque além dos processos normais da tarefa há processos extras para coletar os dados da execução e processos para exibição dos dados para o desenvolvedor.

O tempo para terminar a execução utilizando o *profilers* nativo da JVM foi muito mais longo que os demais, passando dos 60 minutos de execução. Por isto, seus resultados não foram empregados.

O Jprofileré em torno de 1,8x mais rápido do que o Jprobe, entregando no mesmo tempo uma análise idêntica de qual é o método gargalo, este resultado éo esperado ao basear-se em trabalhos correlatos como (ENGEL *et al.*, 2015).

A tabela 2 apresenta a estimativa de quanto tempo de execução da classe *K-means* é gasto em cada método.

Tabela 2 – Métodos

Nome do método	Tempo de execução(s)	Tempo gasto no método(%)
weka.clusterers.SimpleKMeans.moveCentroid	690	50
weka.core.instanciaance.weight	171	12
weka.core.instanciaance.numAttributes	107	7
weka.core.instanciaance.attributes	99,837	7
weka.core.instanciaance.isMissing	88,793	6
weka.core.instanciaance.value	88,499	6
weka.core.Attribute.isNumeric	48,7	4
weka.core.DistanceFunction.distance	37,974	2

Fonte: O autor

Com estes dados pode concluir-se que o gargalo principal do *K-means* no Wekaé o metodo moveCentroid.

A identseicação do gargalo da classe *K-means* feito com *profiler*, abrange apenas o escopo do método que gera o gargalo, um método pode ser extenso, o que gera uma verseicação manual por parte do desenvolvedor para localizar dentro do método quais pontos podem estar ocasionando o gargalo. Porém, ao utilizar um *profiler* existe uma grande possibilidade de diminuir a quantia de código que irá ser verseicado manualmente pelo desenvolvedor.

A classe *K-means* tem 2472 linhas enquanto o método moveCentroidtem 102 linhas, a dseerença entre o escopo que o desenvolvedor precisa verseicar para encontrar o gargalo fica ilustrado no gráfico da figura 8.

Figura 8 – Linhas a serem analisado com e sem o uso de *profiler*

Fonte: O autor

A redução de escopo que será necessário analisar pelo desenvolvedor na figura 8. Notamos que ao utilizar o *profiler* o número de linhas que serão necessárias analisar foi reduzido em 95,57%. Resultado esperado de acordo com o trabalho de (ENGEL *et al.*, 2015).

O Jprofiler é uma ferramenta paga que pode ser obtido com uma licença vitalícia por US$200,00, ou ser utilizado por 10 dias em modo de avaliação.

Com o método computacionalmente mais intenso identseicado, foi feita uma análise *top-down* do método para identseicar o(s) ponto(s) que estaria(m) gerando o gargalo do algoritmo, este método não emprega estrutura de repetição, com exceção das estruturas de repetição transcritas no trecho de código 1:

Codigo fonte 1: Trecho do moveCentroids.

```
.
.
.
// Primeira Regiao
para (instanciaacia    :membro) faca
        para (j; Atributos; j++) faca
                se(instanciaancias.isMissing) faca
                        weightMissing[j]+= instancincias.weight(
                facase senao
                faca
                        weightNonMissing[J] += instanci ncias.weig
```

```
                                se ( membro e numerico ) faca
                                        vals[j]   += instanci ncias.weight
                                facase facasenao

        facapara
facapara


// Segunda regiao
para (j; membros.numAtributtes;  j++) faca
        Se(membros.atributos(j).Numerico,   j++) faca
                Se ( weight .NonMissing [j] > 0 ) faca
                        Vals [j] = weight Non Missing
                facase senao
                        Vals [j] = Utils. misisng Value
                facasenao
        senao faca
                max = -Double . Max
                maxIndex = -1
                para (i;  nominal Dist [j].lenght;i++) faca
                        se ( nominal Dist [j][i] > max) faca
                                max = nominal Dist [j][i]
                                maxIndex = i
                se(max < weightMissing [J])  faca
                                vals[j] = utils.missingValue
                facase senao
                faca
                        vals [J] = maxIndex
                facasenao
    facase facapara
```

O trecho anterior foi organizado para um melhor entendimento da seguinte forma, ele como um todo será referenciado como item 1, o primeiro *loop*, o qual possui outro *loop* em seu interior será referenciado como regiao 1, o terceiro e quarto loop ao ler-se o código de cima para baixo será referenciado como item 2.

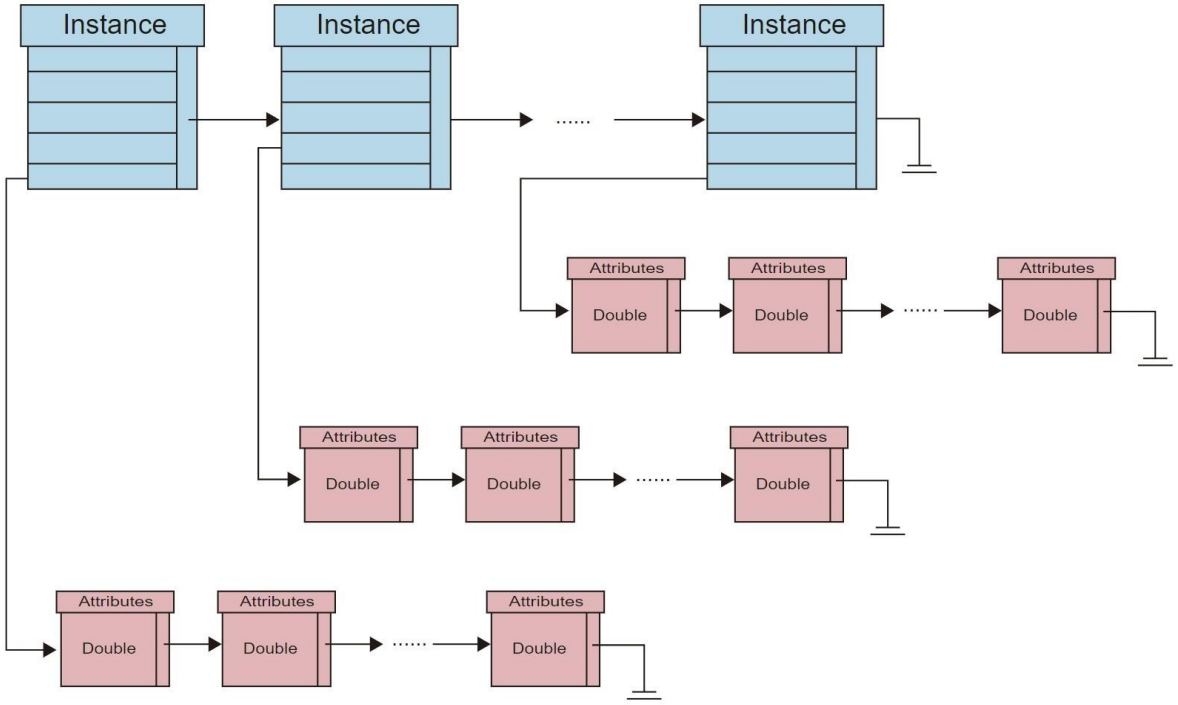
3.4 REESCRITA DO MÉTODO PARA A VERSÃO PA-RALELA

Primeiramente o trecho de código foi encapsulado por uma estrutura de repetição do tipo *IntStream*,

Esta estrutura será a responsável por gerenciar as *threads* que irão executar o item 1 do código. N é o número de *threads* que serão criadas e z é o consumidor do lambda e pode ser entendido como um índice para selecionar determinada *threads*.

Para dar procedimento ao trabalho é necessário um entendimento das estruturas de dados utilizadas pelo método moveCentroids a qual é exemplseicada na figura 9.

Figura 9 – Estrutura de Dados



Fonte: O autor

Existe uma lista de objetos do tipo "instanciaance", estes objetos além de possuírem atributos possuem também uma outra estrutura de dados interna, uma lista do tipo *double* que armazena os atributos da base de dados.

O JIT é incapaz de interagir com objetos e de invocar métodos por isto foi necessário criar estruturas de dados unidimensionais auxiliares para armazenar

os valores dos objetos utilizados nas regiões A e B, além de estruturas para armazenar os valores de retorno dos métodos usados.

Primeiro foi criado um vetor de nome *isMissing* que é um vetor de booleanos com tamanho igual a instanciâncias total da base de dados multiplicado pelo número de atributos, ele será responsável por armazenar se o valor do elemento correspondente por ele está presente ou não, e será utilizado no primeiro item.

O segundo vetor criado foi nomeado como *isNumeric*, é um vetor de booleanos, com tamanho igual ao número de atributos, ele será responsável por armazenar se o valor do elemento correspondente por ele é numerico ou não, e será utilizado na segunda regiao.

O terceiro vetor(de nome *value*) será responsável por armazenar o valor do elemento e terá o mesmo tamanho do vetor *isMissing*, sendo ele um vetor de inteiros, o terceiro vetor(de nome *weight*) é um vetor *double* e armazenará o peso da instanciâncias e seu tamanho será igual ao número de instanciância , a constante (de nome missingValue é utilizada para quando o elemento não tem seu valor definido).

O quarto vetor (de nome *isNumeric*) é um vetor de booleanos que armazenara se o atributo da instanciâncias corrente é numérico ou não. Seu tamanho é igual ao número de instanciância .

O quinto e último vetor(*nominalDistAux*) é mais uma estrutura de dados auxiliar que armazenara em cada *thread* o valor que deveria ser atribuído a matriz *nominalDist*, após a execução do trecho paralelo os valores deste vetor são copiados para a matriz. A seguir a inicialização destas estruturas de dados:

As estruturas de repetição do item 1 e 2 foram alteradas para cada *thread* receber como carga de trabalho uma fração das posições de cada um dos vetores para realizar suas estimativas. Porém, os vetores possuem dois tamanhos dseerentes (o mesmo tamanho dos atributos e o tamanho igual a multiplicação de atributos por instanciância), assim não basta criar um número de *threads* que seja divisor do tamanho de algum dos vetores, pois dseicilmente este número de *threads* poderia dividir de maneira que não houvesse resto os dois vetores.

Para solucionar este problema existem diversas alternativas, algumas delas são: (i) ignorar as posições do resto da divisão, (ii) selecionar alguma *thread* para receber as posições restantes,(iii) re-dividir as posições remanescentes do vetor para serem alocados em algumas *threads*, (iv) utilizar o máximo divisor comum entre o número de posições dos vetores como o número de *threads* para não existir resto entre de posição em qualquer um dos vetores.

A solução (i) tem como problema que ao ignorar posições remanescentes de alguns dos vetores haverá uma queda no resultado qualitativo da versão paralela em relação a sequencial. A solução (ii), geraria um grande desbalanceamento entre as *threads* o que potencialmente causaria queda desempenho, Utilizando a solução (iii) o desbalanceamento entre as *threads* seria potencialmente menor, mas ainda assim afetaria o desempenho, A escolha da solução (iv) ocasionaria em um número normalmente baixo de *threads*, mas garantiria o balanceamento das cargas de trabalho, além da precisão qualitativa da implementação. Neste trabalho optou-se pela solução (iv).

As estruturas de repetição dos itens 1 e 2 foram alteradas para cada *threads* executar o seu segmento das estruturas de dados. Para isto foi criado um método que divide as instanciância e atributos que estão sendo analisados pelo número de *threads*, gerando dois quocientes(a e b), por isto recomenda-se um número de *thread* capaz de dividir ambos elementos sem haver algum resto. O primeiro quociente (a) é referente ao número de instanciância da base de dados carregada e o segundo quociente (b) é referente aos atributos da base de dados, ambos os quocientes são utilizados para definir os limites superiores e inferiores de cada segmento da estrutura de repetição que serão executados por cada uma das *threads*.

A definição dos limites superiores e inferiores foi através da seguinte equações:

$$limiteInferior = (z * y) \tag{3.1}$$

$$limiteSuperior = (z * y) + y \tag{3.2}$$

Onde z é o número de threads e y é o quociente a ou quociente b, dependendo de qual das estruturas de repetição está sendo modseicada. Desta maneira o "para"modseicado para executar apenas o fragmento específico do vetor que cabe a aquela *threads* executar, ficando com a seguinte parama:

Codigo fonte 4: Estrutura para gerenciamento de *Threads*

```
...
faca (a=z*y;(z*y)+y;a++){
...
}
```

O item 1 foi reescrito e ficou da seguinte maneira:

Codigo fonte 5: regiao 1 com os paras modseicados.

```
para (instanciaancia = (N*a);((N*a)+a)     ;instanciaancia++) faca
    para (j=(N*b);((N*b)+b);j++) faca
        se (instanciaancia.Vazia(j))    faca
            weightMissing[j] += instanciaancia.peso();facase
        senao faca
            weightNonMissing[j] += members.get(instancia).weight();
            se (instanciaacia.Numerica)faca
                vals[j] += instanciaancia.Peso * instanciaancia.val
            facase senao {
                nominalDists[j][(int)instanciaancia.valor(j)] +=ins
            facsenao
        facase
    facapara
facapara
```

A seguir foi feita mesma alteração no item 2, como pode ser observado no trecho de código a seguir:

Codigo fonte 6: item 2 com os paras modseicados.

```
para (j = (N*b);((N*b)+b);   j++) faca
    se (membro(j).Numerico())  faca
        se (weight Non Missing[j] > 0) faca
            vals[j]  /= weight Non Missing[j]
        ;facase senao
            vals[j]= Utils.missingValue();
        facasenao
    facase senao faca
        max = -Double.MAX_VALUE;
        maxIndex = -1;
        para(i = 0; nominalDists[j].length; i++){
            se(nominal Dists[j][i] > max) faca
```

```
                    max = nominal Dists [ j ] [ i ] ;
                    maxIndex = i ;
                facase
                se ( max < weight Missing [ j ] ) faca
                    vals [ j ] = Utils . missingValue ( ) ;
                facase senao
                    vals [ j ] = maxIndex ;
                facasenao
            facapara
        facase
    facapara
```

Em sequência, tanto a regiao 1 quanto a regiao 2 tem seus trechos que utiliza- vam chamadas de métodos substituídos pelos vetores acessando as posições equivalentes para aqueles pontos de execução

Codigo fonte 7: inicialização dos vetores.

```
...
para(instanciaancia = (N*a); instancia <((N*a)+a)     ; instancia++) faca
    para (int j = (N*b); j < ((N*b)+b); j++) faca
        se (isMissing[(instancia+1)*j]) faca
            weight Missing [j] += weight [j];
        facase senao faca
        weight Non Missing [j] += weight [j];
        se (is Numeric [j]) faca
            vals[j] += weight [j] * value [(instancia+1)*j];
        facase senao faca
            nominalDistsAux[j * value[((instancia+1)*j)]]    += weight
        facasenao
    facapara
facapara


para (int j = (N*b); j < ((N*b)+b); j++) faca
    se (is Numeric [j]) faca
        se ( weight Non Missing [j] > 0 ) faca
            vals [j] /= weight Non Missing [j]
        ; facase senao faca
            vals[j] = missing Value;
        facasenao
```

```
        facase senao faca
            double max = -Double . MAX_VALUE;
            double maxIndex = -1;
            para (int i = 0; i < nominalDists[j].length; i++) faca
                se (nominal Dists[j][i] > max) faca
                    max = nominal Dists[j][i];
                    maxIndex = i;
                facase
                se (max < weight Missing[j]) faca
                    vals[j] = missing Value;
                facase senao faca
                    vals[j] = maxIndex;
                facasenao
            facapara
        facasenao
    facapara
    ...
```

O método para comparar a versão serial e paralela do algoritmo foi organizado em três etapas: (i) execução da versão serial do algoritmo e seus resultados armazenados. Em sequência será feito o mesmo procedimento para a versão utilizando o ambiente *manycore*, e seus resultados serão comparados para medir a consistência dos resultados da versão paralela com relação a versão serial. (ii) Serão feitas dez medições de tempo para os seguintes casos: 1*thread*, 2*thread*, 4*thread*, 8*thread* e utilizando a GPU. (iii) análise estatista dos tempos obtidos.

Primeiro foram executados os dois algoritmos (versão de CPU e de GPU) para avaliar os resultados qualitativos, quando o número de *threads* é igual ao máximo divisor comum dos dois tamanhos de vetores os resultados são os mesmo por não haver perda de segmento das estruturas de dados. Na tabela a seguir os resultados do algoritmo de agrupamento no caso ideal:

Tabela 3 – Resultados da execução

Cluster	instanciância	Porcentagem
Cluster 0	4400	22%
Cluster 1	6400	32%
Cluster 2	1000	5%
Cluster 3	2800	14%
Cluster 4	2000	10%
Cluster 5	3400	17%

Fonte: O Autor

Na tabela é mostrado como o algoritmo K-means agrupou as amostras da base

dados, o agrupamento foi o mesmo tanto para a versão sequencial quanto para versão paralela

Com a consistência dos resultados conclui-se que a versão paralela do K-means usando o IBM-SDK com auxílio do JIT para quando o número de *threads* é o máximo divisor comum entre os tamanhos dos vetores. O método moveCentroid reescrito para trabalhar em ambiente *manycore*, pode ser encontrado no Apêndice D deste trabalho.

A tabela 4, apresenta os tempos de execução obtidos para a versão puramente sequencial e utilizando a GPU.

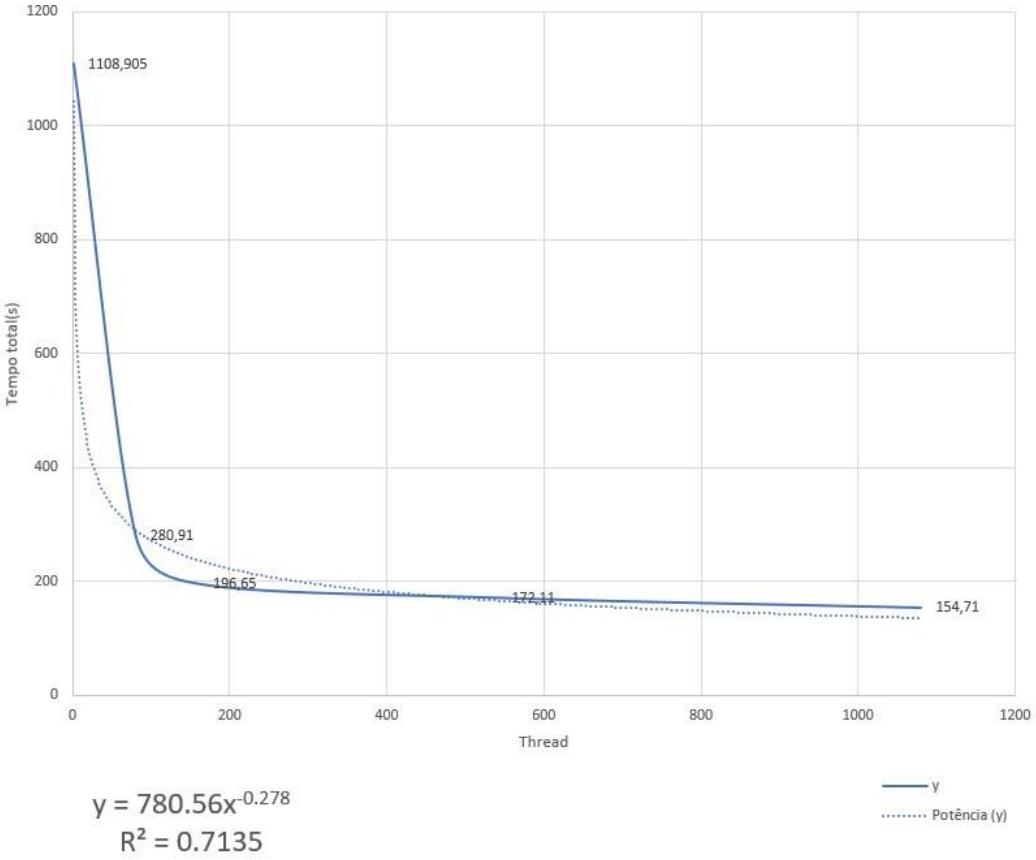
Tabela 4 – Tempos de execução

Número de Threads	1 Threads	80 Threads	160 Threads	540 Threads	1080 Threads
1 execução	00:19:40	00:04:24	00:03:14	00:02:51	00:02:33
2 execução	00:19:08	00:04:32	00:03:17	00:02:57	00:02:34
3 execução	00:18:19	00:04:57	00:03:24	00:03:01	00:02:39
4 execução	00:19:18	00:04:51	00:03:04	00:02:45	00:02:27
5 execução	00:18:30	00:04:30	00:03:12	00:03:06	00:02:36
6 execução	00:18:27	00:04:19	00:03:09	00:02:48	00:02:36
7 execução	00:19:51	00:04:39	00:03:22	00:02:51	00:02:30
8 execução	00:18:27	00:04:48	00:03:21	00:02:49	00:02:31
9 execução	00:17:28	00:04:22	00:03:18	00:02:55	00:02:37
10 execução	00:18:1	00:04:58	00:03:12	00:02:49	00:02:41
Media	00:18:28	00:04:41	00:03:14	00:02:56	00:02:36
Desvio Padrão	0,7	0,23	0,23	0,10	0,13
Coeficiente de Variação	3,842%	4,926%	3,10%	4,627%	2,23%
Speed Up	-	3,926	5,57	6,37	7,09
Eficiência	-	4,88%	3,48%	1,18%	0,65%

Fonte: O autor

No gráfico a seguir podemos observar algumas características do tempo médio de execução neste experimento.

Figura 10 – Tempo médio (S) x Threads



Fonte: O Autor

O ganho na obtenção do tempo de resposta ao utilizar as 80 *threads* foi menor do que o esperado de acordo com resultados da literatura ao utilizar a GPU (NETO, 2014), por isto optou-se por utilizar mais *threads* do que o máximo divisor comum, obtendo assim medidas de desempenho para casos com 160, 540 e 1080 *threads*.

Observa-se que a equação encontrada ao realizar a regressão linear da função do tempo médio por *threads* é um polinômio decrescente, como era o esperado, pois ao aumentar o número de *threads*, a tendencia é diminuir o tempo médio das execuções. A função ficou definida como sendo:

$$y = 780,56x^{-0,278} \tag{3.3}$$

O R no caso do tempo médio ficou definido em 0,7135, ou seja, a função polinomial obtida através da regressão linear consegue explicar 89,91% da função obtida com regressão linear.

Como pode ser observado na linha "Desvio Padrão", o grupo de resultados utilizando 1 *threads* foi o que ficou mais disperso em relação à média, isto ocorreu

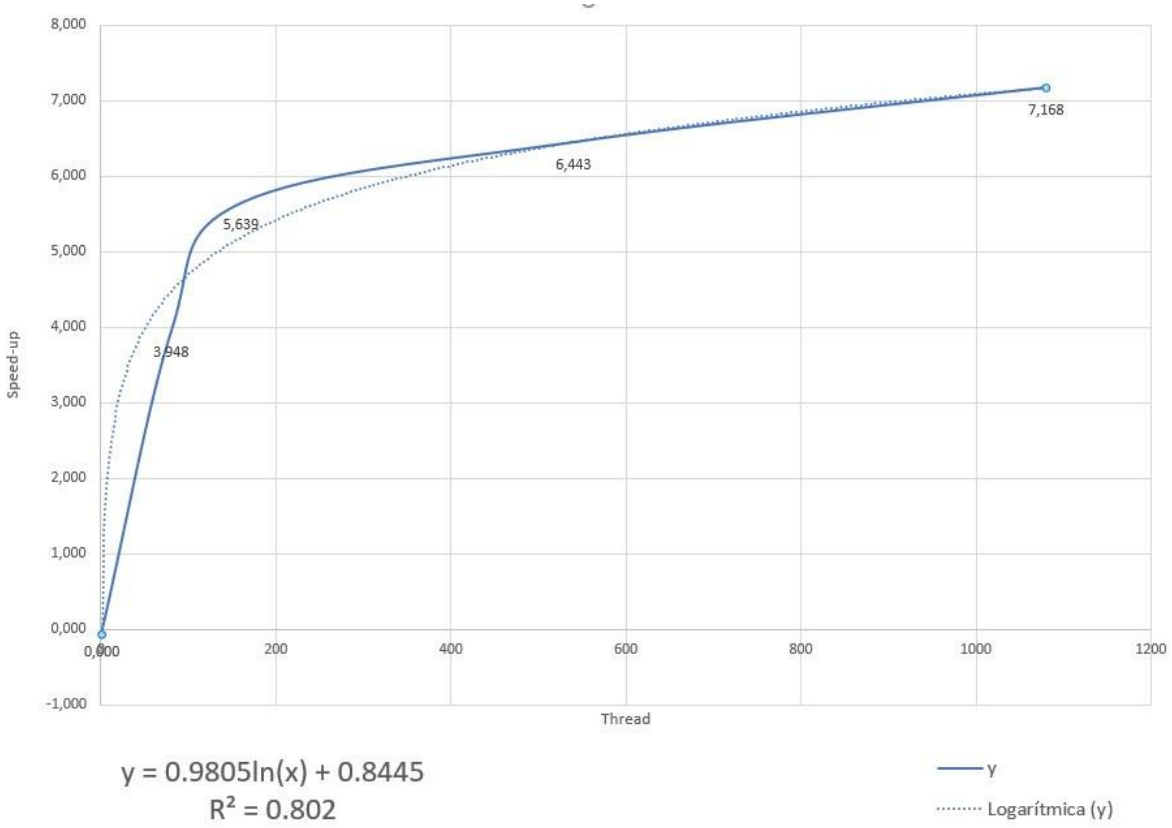
porque é o conjunto com o maior tempo total, ficando em média 18 minutos e 22 segundos executando, sendo então o que está mais suscetível a ter influência externa de algum outro processo durante sua execução e portanto uma dispersão proporcional em relação à média traz um impacto maior neste grupo.

Pode-se ainda observar na linha "Coeficiente de Variação"(CV), que em todos os casos ele o CV foi considerado baixo (abaixo de 15%), isto signseica que os tempos de respostas obtidos em todos os casos foram homogêneos para um determinado número de *threads*.

Concluindo-se que o grupo de 1 *threads* foi o que teve a maior variação em valor absoluto do tempo de execução, enquanto o grupo utilizando a GPU foi o que teve a maior variação proporcional do tempo de execução.

O *Speed Up* de cada um dos casos é apresentado na linha seguinte da tabela, como o esperado o *Speed Up* foi crescente de acordo com o número de *threads*, a figura 11 apresenta um gráfico sobre o *Speed Up*:

Figura 11 – Speed Up x Threads



Fonte: O Autor

Como podemos observar no gráfico 1 há uma crescente no valor do *Speed Up*(eixo y) conforme aumenta-se o número de *threads*, há um queda ao utilizar 8

threads e isto ocorre porque o processador usado têm quatro núcleos, ao utilizar oito núcleos constatou-se um aumento considerável na temperatura, além de ser preciso escalonar entre as *threads* acarretando em queda de desempenho.

No experimento utilizando a GPU, há um crescente que parece ser linear novamente. Porém, deve-se salientar que o número de *threads* cresceu muito em relação as execuções com CPU, ou seja, entre a medição com 8 e com a GPU há um grande intervalo.

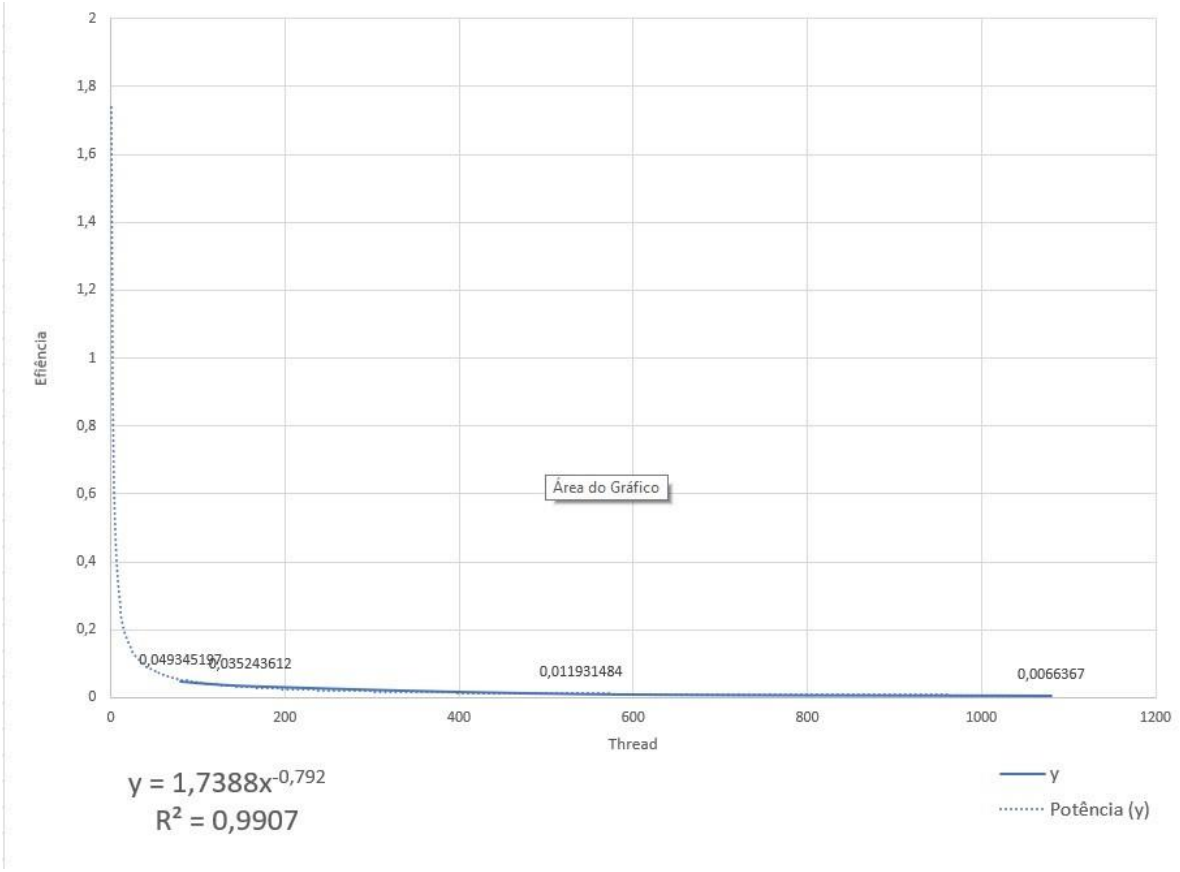
Ao linearizar a função do Speed Up por *threads*, foi encontrado uma função logarítmica, mais especseicamente a função:

$$y = (0{,}9805 * ln(x)) + 0{,}8445 \tag{3.4}$$

Ao ser uma equação logarítmica possui uma assintota horizontal, ou seja, um limite para o valor máximo possível de obter-se com esta esta função. Isto é exatamente o esperado ao paralelizar algoritmos, principalmente como a Lei de Amdhal prevê, que o Speed Up máximo é limitado pelo trecho sequencial do algoritmo que foi paralelizado, sendo este tempo serial a assíntota horizontal da função do Speed Up neste caso.

O R desta função foi de 0,802. O que corresponde a dizer que a função logarítmica consegue explicar 80,2% da função de *Speed Up*. A linha seguinte da tabela 1 foi referente as eficiências dos testes, as quais podem ser observadas na figura 13:

Figura 12 – Eficiencia x Threads



Fonte: O Autor

Apesar da implementação utilizando a GPU ser a que obteve o menor tempo de resposta e o maior *Speed Up*, ela foi a que obteve a pior eficiência, isto pode ser explicado porque o número de Threads aumentou muito em relação a implementação utilizando somente a CPU, o *Speed Up* por outro lado não acompanhou da mesma maneira, isto fez com que a eficiência caísse em relação a implementação com a GPU.

A regressão linear da função de eficiência obteve uma equação decrescente, como o esperado devido ao *Speed Up* não ser linear, a equação é polinomial com um coeficiente negativo, assim conforme o número de *Threads*, aumenta a eficiência decai. Assim como no gráfico do Speed Up, podemos ver uma estabilização na queda de ganho ao aumentar o número de *threads* ao utilizar a GPU, assim como temos novamente uma assíntota horizontal limitando a função, mas desta vez limitando a queda de desempenho.

$$y = 17388^{-0,792} \tag{3.5}$$

O R no caso da eficiência ficou definido em 0,9907, ou seja, a função polino- mial obtida através da regressão linear consegue explicar 99,5% da função da eficiência.

No caso utilizando todos os núcleos da GPU o *Speed Up* foi de apenas 7,09, um ganho pequeno ao comparar com resultados encontrados na literatura (NETO, 2014) conclui em seu estudo que ao aumentar o tempo expendido na regiao paralelizada do *software* melhores resultados foram obtidos utilizando ambiente *manycore*, isto é esperado também pela lei de Amdhal.

Conclui-se que neste caso de estudo o ganho na obtenção do tempo de resposta está sendo limitado pela fração serial do algoritmo. Para confirmar este fato a base de dados foi transformada tanto em número de instanciância quanto em número de atributos, as medições dos tempos de resposta e suas análises encontram-se na tabela 5.

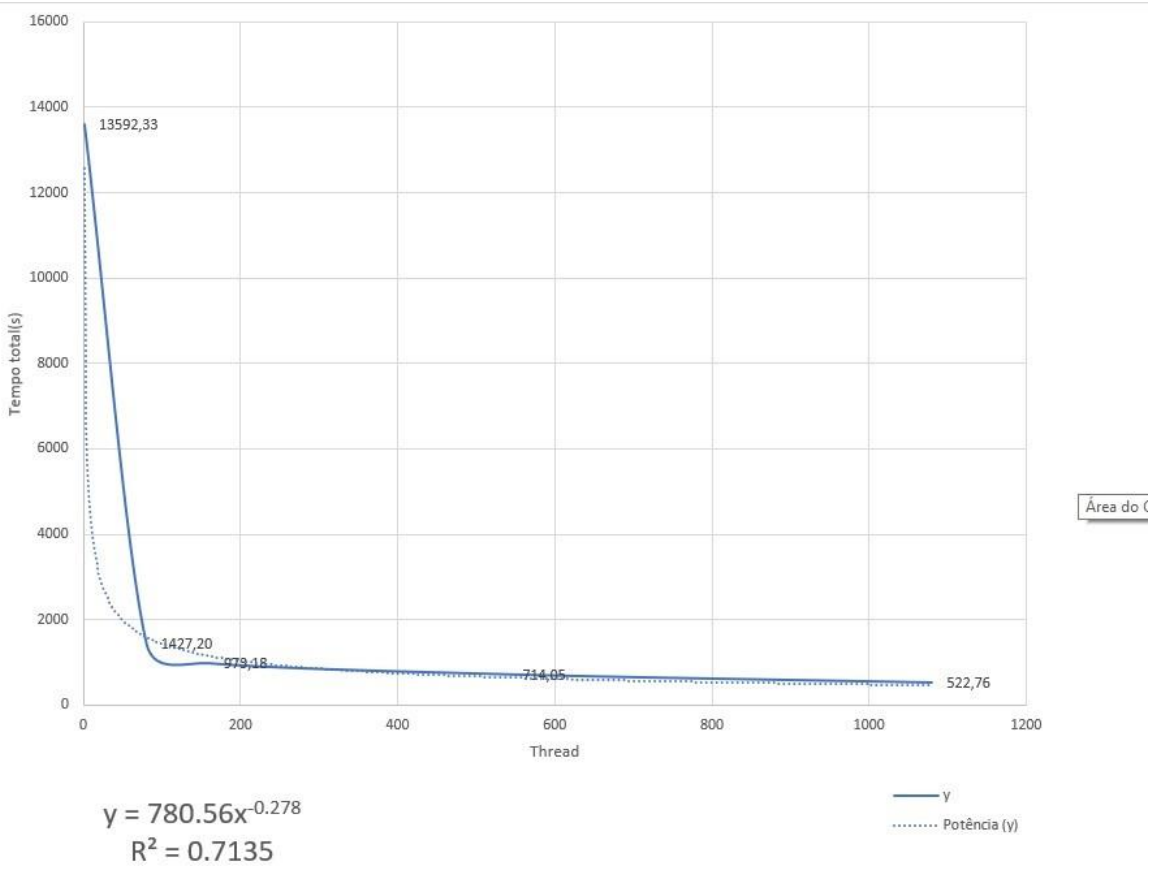
Tabela 5 – Tempos de execução

Numero de Threads	1 Thread	80 Threads	160 Threads	540 Threads	1080 Threads
1 execução	03:46:15	00:23:42	00:16:12	00:12:05	00:08:49
2 execução	03:47:21	00:23:57	00:16:06	00:12:01	00:08:37
3 execução	03:47:18	00:24:57	00:16:04	00:11:49	00:08:39
4 execução	03:47:23	00:24:18	00:16:17	00:11:48	00:08:44
5 execução	03:46:24	00:23:41	00:16:36	00:11:43	00:08:38
6 execução	03:46:36	00:23:21	00:16:06	00:11:59	00:08:45
7 execução	03:46:26	00:23:30	00:16:36	00:12:09	00:08:36
8 execução	03:46:06	00:23:41	00:16:29	00:11:42	00:08:32
9 execução	03:45:54	00:23:48	00:16:12	00:12:17	00:08:37
10 execução	03:47:24	00:23:54	00:16:04	00:11:45	00:08:42
Media	03:46:32	00:23:55	00:16:12	00:11:54	00:08:41
Desvio Padrão	0,596	0,23	0,24	0,20	0,18
Coeficiente de Variação	0,26%	1,010%	1,28%	1,58%	0,75%
Speed Up	-	9,52	13,96	19,03	26,001
Eficiência	-	11,9%	8,72%	3,52%	2,4%

Fonte: O autor

A figura 13 contém características do tempo médio de execução para o experimento com a base de dados ampliada.

Figura 13 – Tempo médio (M) x Threads



Fonte: O Autor

O comportamento da redução do tempo de resposta foi similar ao primeiro experimento, ao utilizar a GPU com a base de dados maior a redução na obtenção do tempo de resposta. Este fenômeno ocorre porque ao aumentar o tempo total de execução que o trecho paralelizado é utilizado, diminui-se o tempo de resposta (NETO, 2014).

A equação obtida ao realizar a regressão linear neste caso foi uma potência decrescente, conforme o esperado pois ao aumentar o número de *threads*, a tendência é diminuir o tempo de resposta médio das execuções, a função foi definida como:

$$y = 780.56x^{-0,278} \tag{3.6}$$

O R no caso do tempo médio para a base de dados maior foi definido como 0,7135, ou seja, a potência obtida através da regressão linear consegue explicar 96,83% da função do tempo total.

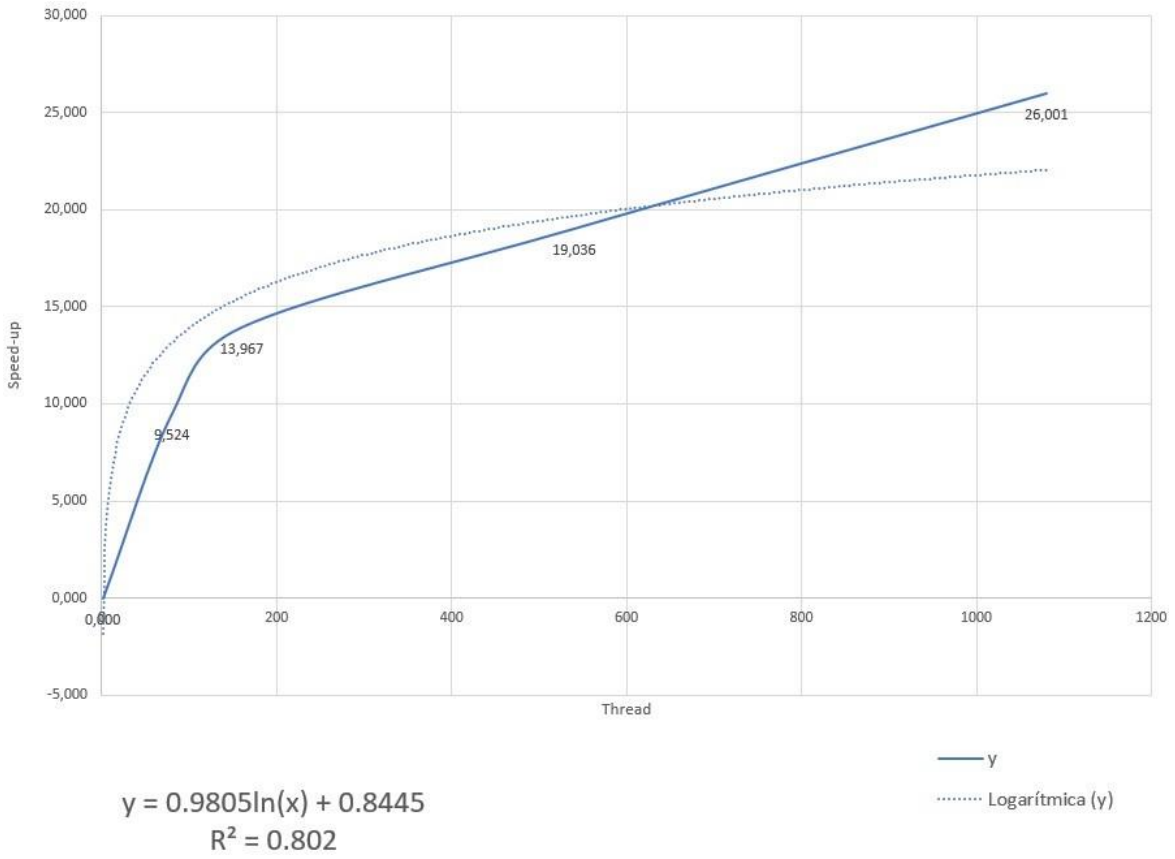
Na linha "Desvio Padrão", o grupo com o menor desvio padrão foi ao utilizar a GPU com 1080 *threads*, isto ocorre porque foi o grupo com o menor tempo de

resposta, o que implica em valores absolutos menores na oscilação, tornando ele o grupo com menor suscetibilidade a sofrer influência externa de algum outro processo.

Na linha "Coeficiente de Variação", que o CV foi considerado baixo (15%), isto significa que os tempos de respostas obtidos em todos os casos foram homogêneos para um determinado número de *threads*. Sendo o conjunto de resultados para execução com uma *thread* o mais homogêneo isto ocorre porque é um conjunto que não precisa haver escalonamento de *threads* tornando assim o que possui a menor variação proporcional de tempos de resposta.

O *Speed Up* de cada um dos casos é apresentado na linha seguinte da tabela, como o esperado o *Speed Up* foi crescente de acordo com o aumento do número de *threads*, a seguir temos um gráfico com mais detalhes sobre o*Speed Up*.

Figura 14 – Tempo médio (M) x Threads



Fonte: O Autor

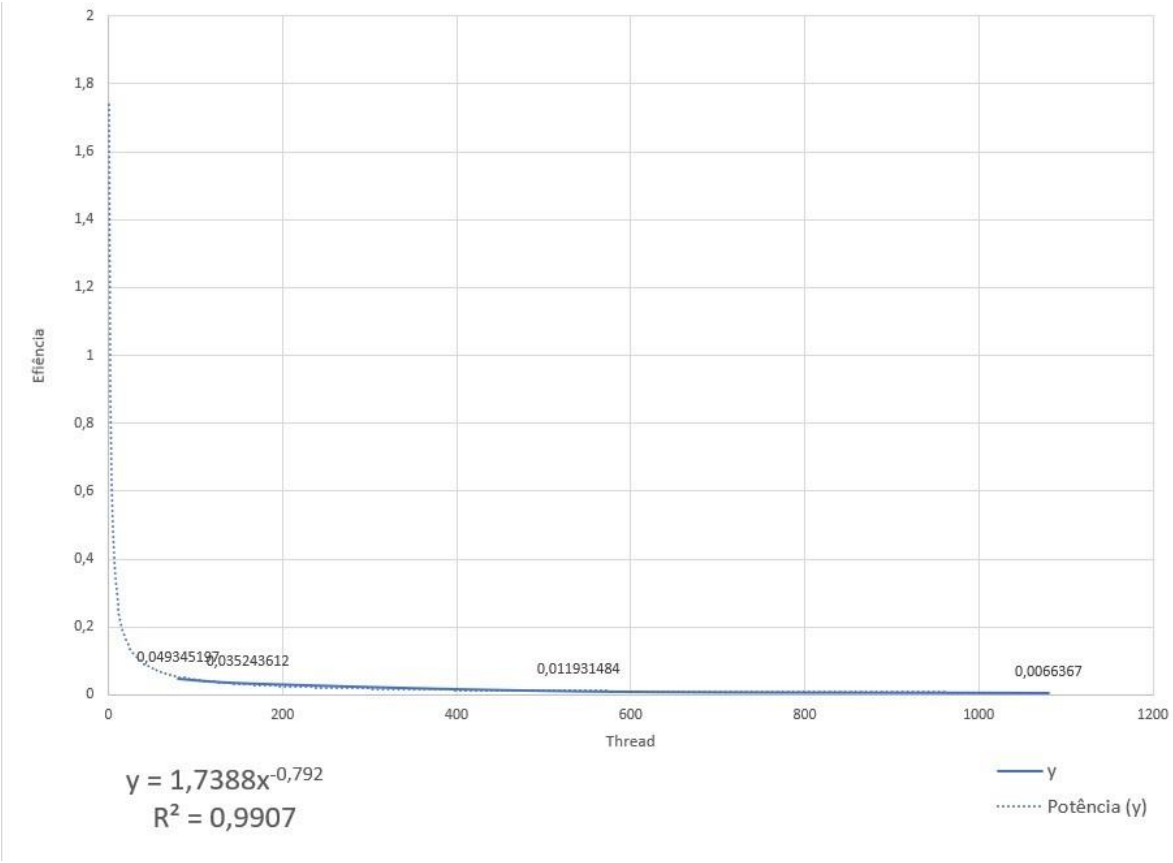
Ao utilizar-se a GPU com a base de dados maior o*Speed Up* é mais robusto assemelhando a uma crescente linear neste experimento. Porém, ele é melhor descrito como uma equação logarítmica, o esperado segundo a lei de Amdhal o *Speed Up* logo será limitado pela fração sequencial do software como um

limitador a assíntota da equação. A função obtida ao realizar a regressão linear foi

$$y = 0{,}9805 ln(x) + 0{,}8445 \tag{3.7}$$

Seu R foi definido como 0,802, o que significa que a equação encontrada com a regressão linear consegue explicar 80,2% da função do *Speed Up*. A linha seguinte da tabela x foi referente a eficiência dos testes com a base de dados maior, e são melhores explanados no gráfico a seguir:

Figura 15 – Tempo médio (M) x Threads



Fonte: O Autor

A eficiência ao utilizar uma base de dados maior, também aumentou em relação ao primeiro experimento deste trabalho, um efeito colateral do aumento do *Speed Up* por aumentar o tempo de execução da parte paralela do algoritmo. A melhora na eficiência em relação ao primeiro caso pode ser concluída com o expoente maior, o que indica um fator decrescente menor ao longo do gráfico. O R quadrado da eficiência foi definido como 0,9907, significando assim que a função encontrada ao realizar a regressão linear explica o gráfico com 99,07% de exatidão.

4 CONCLUSÕES

É essencial que algoritmos tenham o menor tempo possível sem comprometer a confiabilidade dos resultados, isto está tornando-se cada vez mais desafiador com o crescimento de volume das bases de dados que precisam ser analisadas, Uma solução como a deste trabalho importante para que pesquisadores sejam capazes de obter os dados importantes para sua pesquisa.

O objetivo geral deste trabalho foi alcançado ao obter-se uma versão paralelizada do *K-means* que é executada em ambiente *manycore*. Seus objetivos específicos também todos alcançados: Ao utilizar um ambiente *manycore* foi possível obter em um tempo menor os resultados do algoritmo o que diminui o tempo para poder concluir pesquisas relacionadas a agricultura e assim obter tecnologias novas em um menor espaço de tempo. Baseando-se nos resultados obtidos neste trabalho a melhor alternativa para diminuir o tempo de resposta, é a escolha correta do *hardware* sendo ele o principal fator a impactar no desempenho, não necessáriamente trocando ele, apenas selecionando a CPU ou a GPU de acordo com a situação, sendo a CPU indicada para casos onde o nível de paralelização do algoritmo é baixo, e a GPU indicada para casos onde o algoritmo pode ser altamente paralelizado. Foi possível demonstrar que ao tornar possível um código ser executado na GPGPU ao invés de somente ser executado na CPU pode trazer um ganho de desempenho com alteração apenas a nível de software.

O CUDA é uma API que pode ser considerada de baixo nivel atualmente por exigir vasto conhecimento por parte do programador, tanto do algoritmo que deverá ser paralelizado, quanto do próprio CUDA, isto vem na contra mão das APIs atuais que cada vez mais querem abstrair conceitos básicos e tornar o desenvolvimento o mais rápido possível. Essa é uma limitação que a própria Nvidia reconhece e por isto concebeu ferramentas para que desenvolvedores ao redor do mundo pudessem criar compiladores que abstraiam conceitos básicos do CUDA. Um dos objetivos específicos deste trabalho foi desenvolver a solução paralela do K-means utilizando algumas destas ferramentas para abstração do desenvolvimento com CUDA.

A ferramenta para esta abstração foi o IBM-SDK com uso do JIT, por mais que o JIT tenha conseguido dar suporte para o desenvolvimento deste trabalho, também ser a ferramenta mais avançada no sentido de abstração de conceitos básicos do CUDA e ser de fácil utilização ela ainda é uma ferramenta limitada,

inclusive afetando desempenho de softwares que o utilizem para obter ganho no paralelismo. Entre as principais limitações podem ser ressaltadas a incapacidade de trabalhar com métodos e com objetos, o que no JAVA pode ocasionar em uma grande reescrita de trechos de código e novas estruturas de repetição.

O JIT seja é a ferramenta com maior nível de abstração ela possui uma longa documentação, que pode forçar o desenvolvedor a ficar muito tempo preso a especializar-se na utilização do JIT, além disto faz-se necessário salientar que o tempo de inicialização com ele é bem alto e que para utilizá-lo o usuário forca-se a utilizar uma JVM especifica da IBM, ao invés de utilizar a de sua preferência.

Os dados obtidos com o *speed up* são os esperados de acordo com a literatura, uma crescente no ganho de desempenho que torna-se limitado pelo tempo serial do algoritmo. O número maior de núcleos da GPU foi capaz de obter ganho de desempenho maior do que a CPU.

Aumentar a base de dados trouxe um aumento no tempo de execução, ao dobrar o tamanho do K e N, o K-means aumentou em média 12,5 vezes, valor esperado pelo K-means ter uma ordem de grandeza K x N. A crescente no tempo de execução ao ampliar a base dados gerou uma parcela de tempo possível de paralelizar maior, refletindo em resultados melhores pelos indicadores de desempenho como o *Speed Up*, a eficiência continua a não apresentar bons resultados devido ao massivo número de núcleos empregados na solução *manycore*, não conseguir ter um *Speed Up* que corresponda a estes números de núcleos. Parte desse fato, pode também ser atribuído ao baixo valor de *clock* dos núcleos de uma GPU.

4.1 TRABALHOS FUTUROS

Como trabalhos futuros sugere-se investigar a escalabilidade desta solução. Investigar se ao manter o mesmo tempo de processamento utilizando a versão do K-means com processamento em GPU, os dados terão uma acurácia maior por explorar por mais iterações o espaço amostral. Investigar o quão impactante no erro é utilizar um número de *threads* diferente do recomendado e soluções para amenizar ou extinguir o problema.

REFERÊNCIAS

AMDAHL, G. M. Validity of the single processor approach to achieving large scale computing capabilitie. In: IBM. **AFIPS Conference Proceedings**. [S.l.]: AFIPS spring joint computer conference, 1967.

BEKKERMAN, R.; BILENKO, M.; LANGFORD, J. Scaling up machine learning parallel and distributed approaches. **Cambridge University Pressl**, n. 2, 2012.

BORTOLOSI, H. J. **Cálculo Diferencial a Várias Variaveis**. [S.l.]: Edições Loyola, 2002.

BURKE, E.; TILLY, J. **Ant: The Definitive Guide**. [S.l.]: O'Reilly Media, 2002.

CATMULL, E. **A Subdivision Algorithm for Computer Display of Curved Surfaces**. Tese (Doutorado) — University of Utah, Salt Lake City, UT, 7 1974.

DEAN, J. **Big data, data mining, and machine learning: value creation for business leaders and practitioners**. [S.l.]: John Wiley & Sons, 2014.

DUNN, J. C. Well-separated clusters and optimal fuzzy partitions. **Journal of cybernetics**, Taylor & Francis, v. 4, n. 1, p. 95–104, 1974.

ENGEL, T. A. *et al*. Performance improvement of data mining in weka through multi-core and gpu acceleration: opportunities and pitfalls. **Journal of Ambient Intelligence and Humanized Computing**, v. 6, n. 4, p. 377–390, Aug 2015.
ISSN 1868-5145. Disponível em: <https://doi.org/10.1007/s12652-015-0292-9>.

FAYYAD; PLATETSKY-SAPHIRO; SMYTH. From data mining to knowledge discovery: an overview. **Advances in knowledge discovery and data mining**, v. 24, p. 01–34, 1996.

FOLEY, J. A. *et al*. Global consequences of land use. **Science (New York, N.Y.)**, v. 309, p. 570–4, 08 2005.

HAN, J.; PEI, J.; KAMBER, M. **Data mining: concepts and techniques**. [S.l.]: Elsevier, 2011.

HAZELL, P. B. **The Asian green revolution**. [S.l.]: Intl Food Policy Res Inst, 2009. v. 911.

HERRERO, M. *et al*. Smart investments in sustainable food production: Revisiting mixed crop-livestock systems. **Science (New York, N.Y.)**, v. 327, p. 822–5, 02 2010.

HOLMES, G.; DONKIN, A.; WITTEN, I. H. Weka: a machine learning

workbench. In: . [S.l.: s.n.], 1994. p. 357–361.

KIRK, D.; HWU, W.-M. **Programando para processadores paralelos: uma abordagem prática à programação de GPU**. [S.l.]: Elsevier Brasil, 2010.

KUMAR, P.*et al.* High performance data mining using r on heterogeneous platforms. In: **2011 IEEE International Symposium on Parallel and Distributed Processing Workshops and Phd Forum**. Xangai, China: [s.n.], 2011.

MALTHUS, T. R. **An Essay on the Principle of Population..** [S.l.: s.n.], 1872.

MCCALLUM, A.; NIGAM, K.; UNGAR, L. H. Efficient clustering of high-dimensional data sets with application to reference matching. In: CITESEER. **Proceedings of the sixth ACM SIGKDD international conference on Knowledge discovery and data mining**. [S.l.], 2000. p. 169–178.

MILLS, R. T. *et al.* Cluster analysis-based approaches for geospatiotemporal data mining of massive data sets for identification of forest threats. **Procedia Computer Science**, Elsevier, v. 4, p. 1612–1621, 2011.

NETO, C. B. **SIMULAÇÃO CLIMATICA DE DADOS DE VENTO DE REDES P2P UTILIZANDO GPU**. Dissertação (Mestrado) — Universidade Estadual de Ponta Grossa, The address of the publisher, 3 2014.

PÉREZ, M. S. *et al.* Adapting the weka data mining toolkit to a grid based environment. In: SZCZEPANIAK, P. S.; KACPRZYK, J.; NIEWIADOMSKI, A. (Ed.).
Advances in Web Intelligence. Berlin, Heidelberg: Springer Berlin Heidelberg, 2005. p. 492–497. ISBN 978-3-540-31900-9.

REN, B.; AGRAWAL, G. Compiling dynamic data structures in python to enable the use of multi-core and many-core libraries. In: IEEE. **2011 International Conference on Parallel Architectures and Compilation Techniques**. [S.l.], 2011. p. 68–77.

RICKLEFS, R. E. **The Economy of Nature**. [S.l.]: W. H. Freeman, 2010.

ROBERT, L. E. **Lectures on Economic Growth**. [S.l.]: W. H. Freeman,

2004.

SAKR., S.; GABER, M. **Highly Parallel Computing**. [S.l.]: Large Scale and Big Data: Processing and management, 2014.

SANDERS, J.; KANDROT, E. **CUDA by Example: An Introduction to General-Purpose GPU Programming**. [S.l.]: Addison Wesley, 2004.

SCHADT, E. E. Computational solutions to large-scale data management and analysis. **The name of the journal**, n. 2, 2010.

SENGER, L. J. **Escalonamento de processos: uma abordagem**

dinâmica e incremental para a exploração de características de aplicações paralelas. Tese (Doutorado), 2005.

TAGARAKIS, A. *et al.* Management zones delineation using fuzzy clustering techniques in grapevines. **Precision Agriculture**, Springer, v. 14, n. 1, p. 18–39, 2013.

VELOSO, L. H. L. **Algoritmo K-means Paralelo Baseado em Hadoop-Mapreduce para mineiração de dados Agricolas**. Dissertação (Mestrado) — Universidade Estadual de Ponta Grossa, Brasil, 2015.

VIEIRA, M. A. *et al.* Object based image analysis and data mining applied to a remotely sensed landsat time-series to map sugarcane over large areas. **Remote Sensing of Environment**, Elsevier, v. 123, p. 553–562, 2012.

WITTEN, I. H.; FRANK, E. **Hadoop: The Definitive Guide**. 4. ed. O'Reilly, 2015. Disponível em: <https://www.safaribooksonline.com/library/view/ hadoop-the-definitive/9781491901687/>.

WITTEN, I. H. *et al.* **Data Mining: Practical machine learning tools and techniques**. [S.l.]: Morgan Kaufmann, 2016.

Apêndices

APÊNDICE A – CONFIGURAÇÃO DE AMBIENTE WINDOWS

Primeiramente, faça o download do código fonte do weka, o qual é possível encontrar no endereço a seguir: https://github.com/Waikato/weka-3.8

Em sequencia é necessário a instalação do Ant para ser possível compilar o código fonte do Weka para o ".jar". Para isto parte-se do principio que o Java esteja corretamente instalado no ambiente de trabalho. O apêndice não irá cobrir a instalação e configuração para o Java no Windows, pois é um processo puramente automático sem precisar alterar algo em relação a instalação expressa.

Faça o download, do Ant apache no seguinte endereço:

https://ant.apache.org/bindownload.cgi

Após isto extraia a pasta do Ant do arquivo ".rar", a pasta pode ser colocada em qualquer diretório do Windows. Porém, algumas instruções a seguir irão ser feitas partindo de que o Ant foi colocado no diretório "C".

Após a extração da pasta deve ser configurada a variável de ambiente "ANT_HOME", para isto vá para o painel de controle do Windows » Sistema » Opções » avançadas do Sistema, irá abrir uma nova janela, nessa nova janela clique em variáveis de ambiente. Abrirá uma nova janela, nesta janela pode-se adicionar novas variáveis ao ambiente. No campo nome defina a variável como "ANT_HOME", no campo valor especifique o local onde extraiu o ant, neste caso "C:\Ant", dando sequencia a pasta "bin"do ant deve ser adicionada à variavel "PATH", para isto selecione a variavel Path , clique em editar, e adicione a linha a seguir "%ANT_HOME%\bin".

Verifique se a variável "JAVA_HOME"esta devidamente configurada e definida no "PATH", caso não, crie a variável, clicando em nova e definindo o nome dela como "JAVA_HOME"e seu valor sendo definido como o caminho para a pasta do "JDK", por padrão ficaria "C:\Program Files\Java\jdk1.8.0_65", após isto tem de adicionar a pasta "bin"do java ao "PATH", selecionando ela e clicando editar e adicionando a linha "%JAVA_HOME%\bin".

Após isto recomenda-se realizar reboot no sistema. Para ter certeza que o ant e o java estão sendo devidamente reconhecidos após esta configuração inicie o "CMD"do windows e digite o seguinte comando: "java -version", se tudo estiver configurado corretamente irá ser exibido na tela do prompt de comando a versão

instalada do java, em sequencia verifique se o ant esta devidamente configurado ao digitar o comando: "ant -version"se tudo estiver configurado corretamente irá ser exibido na tela do prompt de comando a versão instalada do ant.

Com apenas estes dois passos já possível compilar o código fonte do weka para um arquivo executável (".jar") atraves do prompt de comando, para isto basta navegar para o diretório, no qual o weka foi extraído, dentro da pasta "weka-mater"vai ter o diretório weka, ao estar nele via prompt basta usar o seguinte comando "ant exejar"e o ant irá fazer todo o trabalho de construir o weka, seguindo as instruções pré-definidas no xml, "xml.build".

No entanto para ser capaz de utilizar uma IDE para compilar o weka são necessários ainda mais algumas configurações e programas, como veremos a seguir. Para que o netbeans seja capaz de abrir o projeto do weka, precisa que a ferramenta Apache Maven esteja instalada, o download dela pode ser feita no link a seguir: https://maven.apache.org/download.cgi

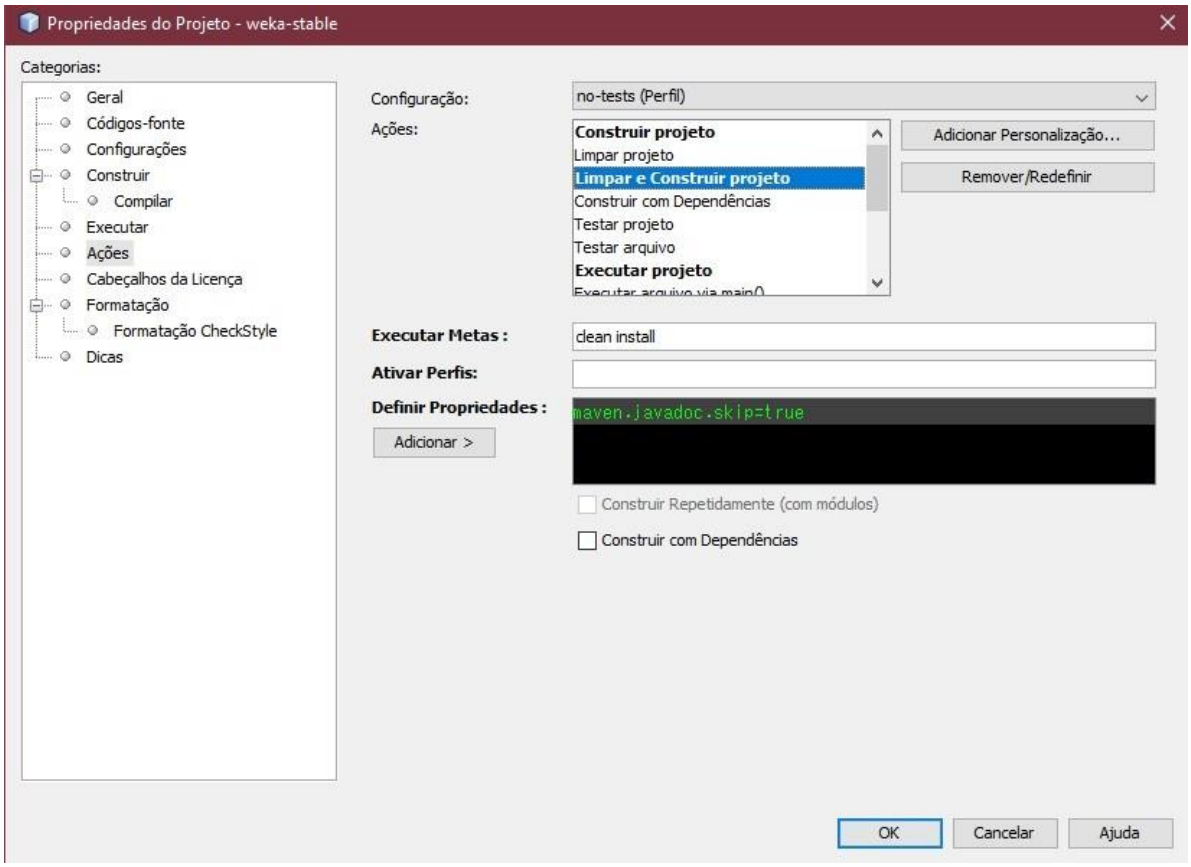
Depois de baixar e extrair o maven basta criar a variável de ambiente "M2_HOME", seguindo o procedimento que foi usado na criação das variáveis de ambiente anteriores, o nome desta nova variável de ambiente deve ser "M2_HOME"e seu valor, o caminho para a pasta que o vane foi descompactado, neste exemplo "C:\Maven". Em sequencia a variável de ambiente precisa ser adicionada no "PATH"do sistema, selecione a variável "PATH"e após isto, clique em editar por fim adicone a seguinte linha "%M2_HOME%"\bin", recomenda-se nesse momento realizar um reboot no sistema.

Para ter certeza que o Mave foi instalado corretamente basta abrir o prompt de comando e digitar o comando a seguir: "mvn –version", se tudo estiver correto irá ser mostrado na tela do prompt a versão instalada do maven.

Nest ponto o netbeans já é capaz de reconhecer o projeto do weka e abri-lo. Porém, ainda precisa-se de algumas configurações no próprio netbeans para que seja possível compilar o código fonte do weka.

Com o Netbeans aberto, selecione "abrir projeto", navegue ate o diretório "weka", que esta dentro do diretório "weka-master"para importar o projeto do weka no netbeans, após isto clique com o lado direito do mouse sobre o projeto e selecione "resolver dependências", o Netbeans irá automaticamente resolver todas as dependências faltantes. A seguir, clique com o lado direito do mouse novamente e selecione a opção "configurações", selecione "Ações", nesta tela terá uma lista de ações, selecione "limpar e construir", e adicionar na caixa "definir propriedades"o seguinte comando "maven.javadoc.skip=true". Ficando como na imagem a seguir.

Figura 16 – Configuração para compilação do Weka atraves do Netbeans



Fonte: O autor

Caso o Netbeans precise que é a classe principal do weka seja especificada para compilar, ela é a classe "Weka.gui.GUIChooser".

APÊNDICE B – CONFIGURAÇÃO DE AMBIENTE LINUX

Primeiramente o IBM-SDK deve estar instalado e configurado corretamente, o download dele pode ser feito no endereço a seguir:
https://developer.ibm.com/javasdk/downloads/

Na versão para linux ele irá vir com a extensão ".bin", para instalá-lo basta usar o comando "sudo ./nome_do_arquivo"e seguir os passos da instalação expressa. Por padrão a instalação será feita no diretorio: "/opt/IBM/java-x86_64-80". Ainda deve-se setar a JAVA_HOME, para isto basta criar um arquivo de texto com o nome enviroment no diretório "/etc"e adicionar a linha: JAVA_HOME="/opt/ibm/java-x86_64-80", em seguida deve-se adicionar a variável de ambiente JAVA_HOME no path, adicionando o trecho a seguir no PATH, "$JAVA_HOME/bin:". E após isto executar esse arquivo com o comando source, da seguinte maneira: "$ source environment". Ainda assim o comando "java"não é reconhecido pelo sistema, precisando fazer uma instalação personalizada do java também especificando onde está a jdk, para ter certeza que o java esta corretamente instalado, basta digitar no terminal o seguinte comando "$ java -version", se tudo estiver certo será mostrado a versão do java e mais alguns detalhes que aparecem a oter o IBM-JDK sendo usado como a opção para o JDK, sendo mostrado no padrão a seguir:

```
java version "1.8.0_191"
Java(TM) SE Runtime Environment (build 8.0.5.27 - pxa6480sr5fp27-20190104_01(SR5 FP27))
IBM J9 VM (build 2.9, JRE 1.8.0 Linux amd64-64-Bit
Compressed References 20181219_405297 (JIT enabled, AOT enabled)
OpenJ9 - 3f2d574
OMR - 109ba5b
IBM - e2996d1)
JCL - 20190104_01 based on Oracle jdk8u191-b26
```

Nota-se que ao utilizar o JDK da IBM, ele também apresenta informação sobre o JIT e se ele esta ativado, isso é imprescindível para utilizar a GPU da maneira que foi feita neste trabalho.

A seguir deve ser feito o download e a configuração do Ant para poder construir o

executável do weka. O download do Ant pode ser feito no link a seguir: https://ant.apache.org/bindownload.cgi

Recomenda-se a instalação manual ao invés de utilizando o comando "$ sudo apt-get ant", pois é imprescindível para a compilação do weka a partir do código fonte que a versão do ant, seja a 1.10.5 ou superior. Após o download e a extração da Ant em algum diretorio, precisa setar a variavel de ambiente "ANT_HOME". Para isto basta abrir novamente o arquivo "enviroment", o qual foi usado para configurar a variavel de ambiente "JAVA_HOME", adicionado o path para a pasta do ant, neste caso a linha ficou como sendo: ANT_HOME="/home/william/Downloads/apache-ant-1.10.5"e adicionado o caminho da pasta "bin"do Ant no PATH do sistema, adicionando o trecho a seguir no valor da variável PATH: "(...)$ANT_HOME/bin:(...)"

Como ultimo passo para ter o Ant definitivamente configurado pasta com o terminal navegar ate a pasta equivalente ao "ANT_HOME", e executar o comando a seguir: "$ ant -f fetch.xml -Ddest=system", com este comando qualquer dependência faltante será baixada.

A seguir será necessário configurar corretamente o CUDA, definir as pendencias e bibliotecas. Como requisitos básicos para rodar programas em CUDA é preciso ter o gcc instalado e alguns kernels. Para verificar se o gcc esta instalado basta digitar na linha de comando do terminal: "$ gcc -version", caso o comando não seja reconhecido basta executar o comando a seguir no terminal: "$ sudo apt-get install gcc". Para instalar os kernels necessários no ubuntu basta executar o comando a seguir no terminal "$ sudo apt-get install linux-headers-$(uname -r)".

O CUDA pode ser baixado no seguinte endereço:
https://developer.nvidia.com/cuda-downloads

Após o termino do download, navegue com o terminal até a pasta aonde o arquivo foi baixado e digite no terminal o seguinte comando:
"sudo sh cuda_10.0.130_410.48_linux.run"

As ações pós instalação devem ser realizadas manualmente, deve-se definir o caminho ate a pasta "bin"do CUDA no PATH do sistema operacional sendo ele por padrão "/usr/local/cuda-10.0/bin". Neste ponto já é possivel executar códigos escritos em CUDA e desenvolver em CUDA, desde que não esteja utilizando alguma biblioteca (Cublas por exemplo). Para poder utilizar todas as bibliotecas do CUDA deve-se adicionar a variável de ambiente "LD_LIBRARY_PATH", para isto basta adicionar a seguinte linha no arquivo "environment": LD_LIBRARY_PATH="/usr/local/cuda-10.0/lib64", após isto o CUDA pode ser plenamente utilizado, para verificar se a configuração esta correta, basta executar a seguinte linha de comando em um terminal:

"$ nvcc –version"

Se estiver tudo configurado corretamente irá ser mostrado a versão do nvcc e do CUDA, além do copyright das tecnologias presentes da Nvidia, ficando similar ao texto a seguir:
nvcc: NVIDIA (R) Cuda compiler driver
Copyright (c) 2005-2018 NVIDIA Corporation
Built on Sat_Aug_25_21:08:01_CDT_2018
Cuda compilation tools, release 10.0, V10.0.130

No entanto para o JIT conseguir utilizar as ferramentas do CUDA acessando a GPU ainda é necessário uma ultima configuração, a de mais uma váriavel de ambiente, que deve ser setada em toda execução via terminal, com o seguinte comando:
"$ export LD_LIBRARY_PATH=$LD_LIBRARY_PATH:$NVVM_LIBRARY_PATH"

Para ter certeza de que esta configuração foi feita de maneira correta, basta executar qualquer comando "java"e tentar acessar a GPU simultaneamente, pode inclusive ser feito apenas pedindo para o Java exibir a versão instalada e habilitar a placa de video simultaneamente e utilizar a opção da JVM "verbose", que é a opção para exibir em tela quando o JIT começa e termina de compilar um método. Pode ainda ser especificado para informar apenas um método que o JIT teve de compilar, neste caso o acesso a GPU. O comando ficaria:
"$ java -Xjit:enableGPU=verbose -version"

Além de apresentar em tela a versão do java deve como resultado printar o dispositivo gráfico, sua versão e seu poder de computabilidade, como a seguir:
"$ [IBM GPU JIT]: Device Number 0: name=GeForce GTX 1060 6GB, Compute-Capability=1024.64"

Quando o IBM-JDK era o JDK selecionado para o sistema trabalhar, tanto o Eclipse quanto o Nebeans (*Standart*), não foram capazes de iniciar corretamente falhando na iniciação dos módulos. Podem ser usados duas soluções: (i) Alterar entre o JDK usado com o comando "$ sudo update-alternatives –config java", o que irá listar todos os JDK instalados, quando for utilizar a IDE selecionar o Oracle JDK ou algum outro que elas consigam iniciar, como o Open JDK, e quando for executar o código realizar o mesmo procedimento para selecionar o IBM-JDK, lembrando que desta maneira não seria possível compilar através das IDEs, seria necessário fechá-las e executar via terminal, e quando fosse voltar a utilizar as IDEs precisaria trocar o JDK de novo.

A outra solução foi utilizar a distribuição do NetBeans da Apache o qual pode ser encontrado para download no endereço a seguir:

https://netbeans.apache.org/download/nb100/nb100.html

Após a instalação, selecione *tools*»*Java Plataforms*»*Add*, em sequencia basta adicionar o caminho ate o IBM JDK. Depois clique com o lado direito do mouse no projeto "weka»>Properties»Compile e selecione para a IDE trabalhar com o JDK 8.

A seguir, instruções necessários para o JIT executar plenamente.

O JIT possui uma extensão documentação a qual pode ser encontrada no seguinte endereço:
https://www.ibm.com/support/knowledgecenter/en/SSYKE2_8.0.0/com.ibm.java.vm.80.doc/docs/jit_overview.html

A seguir serão abordados apenas as opções da dele que foram usadas na JVM:

-Xjit:enableGPU: Opção para especificar que a GPU deve ser utilizada na execução do proximo trecho de código, pode ser usado a opção verbose para verificar se houve a inicialização corretado *device*.

-Xquickstart: Aplicações com um fluxo de dados não tão grande podem não serem otimizadas para utilizar a GPU através do JIT por isto é melhor especificar para os trechos de código paralelizados serem executados na GPU com este comando.

-Xmx6g: Comando que não é especifico do JIT, no entanto foi preciso utilizá-lo para executar o Weka sem problemas decorrentes de falta de memoria.

APÊNDICE C – VERSÃO PARALELIZADA DO MÉTODO MOVECENTROIDS

Codigo fonte : Trecho do movecentroids.

```
protected double[] moveCentroid(int centroidIndex, Instances members,
        boolean updateClusterInfo, boolean addToCentroidInstances) {

    double[] vals = new double[members.numAttributes()];
    double [][] nominalDists = new double[members.numAttributes()][];
    double[] weightMissing = new double[members.numAttributes()];
    double[] weightNonMissing = new double[members.numAttributes()];

    // Quickly calculate some relevant statistics
    for ( int  j  = 0;  j  < members.numAttributes(); j++) {
        if (members.attribute(j). isNominal()){
            nominalDists[j ] = new double[members.attribute(j).numValues
                ()];
        }
    }

    int N = 80;
    int a = this . Quociente(N,members.size());
    int b = this . Quociente(N,members.numAttributes());

    boolean isMissing[] = newboolean[members.size()∗members.
        numAttributes()];
    boolean isNumeric[] = new boolean[members.numAttributes()];
    int [] value = new int[members.size()∗members.numAttributes()];
    double[] weight = new double[members.numAttributes()];
    double[] nominalDistsAux = new double[members.size()∗members.
        numAttributes()];
    double missingValue = Utils.missingValue();
```

```
for ( int x = 0; x < members.size();x++){
    for ( int y = 0; y < members.numAttributes();y++){
        value[( x+1)*y] = members.get(x).numValues();
        isMissing[(x+1)*y] = members.get(x).isMissing(y);

    }
}

 for ( int x = 0; x < members.numAttributes();x++){
     weight[x] = members.get(x).weight();
     isNumeric[x] = members.attribute(x).isNumeric();
 }

IntStream.range(0, N). parallel () . forEach(z -> {
    for ( int inst = (N*a);inst<((N*a)+a) ;inst++) {
        for ( int j = (N*b); j < (( N*b)+b); j++) {
            if (isMissing[( inst+1)*j ]) {
                weightMissing[j] += weight[j ];
            } else {
                weightNonMissing[j] += weight[j ];
                if (isNumeric[j ]) {
                    vals[ j ] += weight[j ] * value[( inst+1)*j ]; // Will
                        be overwritten in Manhattan case
                } else {
                    nominalDistsAux[j * value [(( inst+1)*j) ]] +=
                        weight[j ];
                }
            }
        }
    }

    for ( int j = (N*b); j < (( N*b)+b); j++) {
        if (isNumeric[j ]) {
            if   (weightNonMissing[j] > 0) {
                vals[ j ]  /= weightNonMissing[j];
            }  else {
                vals[ j ]  = missingValue;
            }
```

```
            } else {
                double max = −Double.MAXVALUE;
                double maxIndex = −1;
                for ( int  i  = 0;  i  < nominalDists[j ]. length;  i++) {
                    if (nominalDists[j ][ i ] > max) {
                        max = nominalDists[j][ i ];
                        maxIndex = i;
                    }
                    if (max < weightMissing[j]) {
                        vals[ j ] = missingValue;
                    } else {
                        vals[ j ] = maxIndex;
                    }
                }
            }
        }

}) ;


for ( int i = 0; i < members.size(); i++){
    for ( int j = 0; j < members.numAttributes(); j++){
        nominalDists[i ][ j ] = nominalDistsAux[(i+1)∗j ];

    }
}
// Termina aqui
if (mDistanceFunction instanceof ManhattanDistance) {

    // Need to replace means by medians
    Instances sortedMembers = null;
    int middle = (members.numInstances() − 1) / 2;
    boolean dataIsEven = ((members.numInstances() % 2) == 0);
    if (mPreserveOrder) {
        sortedMembers = members;
    } else {
        sortedMembers = new Instances(members);
    }
    for ( int j = 0; j < members.numAttributes(); j++) {
```

```
                    if (( weightNonMissing[j] > 0) && members.attribute(j).
                        isNumeric()) {
                         // singleton special case
                         if (members.numInstances() == 1) {
                              vals[ j ] = members.instance(0).value(j);
                         } else {
                              vals[ j ] = sortedMembers.kthSmallestValue(j, middle
                                   + 1);
                              if (dataIsEven) {
                                   vals[ j ] = (vals[ j ] + sortedMembers.
                                        kthSmallestValue(j, middle + 2)) / 2;
                              }
                         }
                    }
               }

               if (updateClusterInfo) {
                    for ( int j = 0; j < members.numAttributes(); j++) {
                         mClusterMissingCounts[centroidIndex][j] = weightMissing[j ];
                         mClusterNominalCounts[centroidIndex][j] = nominalDists[j ];
                    }
               }

               if  (addToCentroidInstances)  {
                    mClusterCentroids.add(new DenseInstance(1.0, vals));
               }

               return  vals;
          }
```