UNIVERSIDADE FEDERAL FLUMINENSE

LEONARDO SANTANA VIEIRA

IMPLEMENTAÇÃO PARALELA EM CUDA PARA O ALGORITMO DUAL SCALING EM DADOS DE ORDEM DE CLASSIFICAÇÃO

Niterói

2019

LEONARDO SANTANA VIEIRA

IMPLEMENTAÇÃO PARALELA EM CUDA PARA O ALGORITMO DUAL SCALING EM DADOS DE ORDEM DE CLASSIFICAÇÃO

Trabalho de Conclusão de Curso submetido ao Curso de Tecnologia em Sistemas de Computação da Universidade Federal Fluminense como requisito parcial para obtenção do título de Tecnólogo em Sistemas de Computação.

Orientador:
ALTOBELLI DE BRITO MANTUAN

NITERÓI
2019

Folha reservada para a ficha catalográfica

LEONARDO SANTANA VIEIRA

IMPLEMENTAÇÃO PARALELA EM CUDA PARA O ALGORITMO DUAL SCALING EM DADOS DE ORDEM DE CLASSIFICAÇÃO

Trabalho de Conclusão de Curso submetido ao Curso de Tecnologia em Sistemas de Computação da Universidade Federal Fluminense como requisito parcial para obtenção do título de Tecnólogo em Sistemas de Computação.

Niterói, 06 de dezembro de 2019.

Banca Examinadora:

_____

Prof. Altobelli de Brito Mantuan, MSc. – Orientador

UFF - Universidade Federal Fluminense

_____

Prof. ou Profª. <NOME>, <Título>. – Avaliador

<Sigla da Universidade> - <Nome da Universidade>

Dedico este trabalho à minha mãe Jussara.

AGRADECIMENTOS

A meu Orientador Altobelli de Brito Mantuan pelo estímulo e atenção que me concedeu durante o curso.

Aos Colegas de curso pelo incentivo e troca de experiências.

A todos os meus familiares e amigos pelo apoio e colaboração.

“Quem não sabe o que busca, não identifica o que acha”.

Immanuel Kant

RESUMO

Atualmente vivemos um cenário em que é cada vez mais necessário otimizar a forma como tratamos os dados disponíveis, isto devido ao substancial aumento no volume destes para serem analisados e transformados em informação, por isso se faz necessário o desenvolvimento de novas técnicas mais eficientes para tratar grandes volumes de dados. O *Dual Scaling* é uma dessas técnicas e tem por objetivo mapear itens e transações de uma base de dados como pontos em um espaço multidimensional, onde as distâncias entre os mesmos demonstram o quão relacionados são os itens na base de dados. Entretanto, o modelo matemático utilizado por esta técnica é altamente custoso e o fato de só existirem implementações sequenciais disponíveis na literatura somente amplia este problema. Neste trabalho, é utilizada a plataforma de computação paralela CUDA e a biblioteca Cusp para o desenvolvimento de uma implementação paralela do algoritmo de *Dual Scaling*. Após o detalhamento da solução, são realizados testes comparando o tempo de execução desta solução a uma implementação sequencial em C++ utilizando a biblioteca Eigen e através destes testes é possível verificar a redução do tempo de processamento da implementação proposta.

Palavras-chaves: *Dual Scaling*, CUDA, Cusp, C++ e programação paralela.

ABSTRACT

Currently we live in a scenario where it is increasingly necessary to optimize the way we treat available data, due to the substantial increase in the volume of data to be analyzed and transformed into information, so it is necessary to develop new techniques, more efficient in handling large volumes of data. Dual Scaling is one such technique and aims to map items and transactions from a database as points in a multidimensional space, where the distance between them demonstrates how closely the items in the database are related. However, the mathematical model used by this technique is highly costly and the fact that there are only sequential implementations available on the market only magnifies this problem. In this work, the CUDA parallel computing platform and the Cusp library are used to develop a parallel implementation of the Dual Scaling algorithm. After detailing the solution, tests are performed comparing the execution time of this solution to a sequential implementation in C ++ using the Eigen library and through these tests it is possible to verify the reduced processing time of the proposed implementation.
.

Key words: Dual Scaling, CUDA, Cusp, C++ e parallel programming.

LISTA DE FIGURAS

LISTA DE TABELAS

SUMÁRIO

1 INTRODUÇÃO

Os dados hoje em dia são produzidos constantemente e em volumes cada vez maiores, empresas dos mais diversos ramos trabalham intensamente com grandes quantidades de dados gerados a partir de transações de seus negócios. E uma empresa para se manter competitiva precisa buscar formas de analisar esses dados e produzir informações úteis a seu negócio da forma mais eficiente possível.

Para atender a essa necessidade, são necessárias ferramentas que sejam capazes de processar esse volume crescente de dados. Essas ferramentas estão em constante evolução, sempre surgindo novas ou aprimorando as existentes através do desenvolvimento e implementação de novas técnicas, algoritmos e modelos matemáticos, sempre com o objetivo de se processar os dados da forma mais eficiente possível.

Nesse cenário, os modelos matemáticos são de especial importância, devido a capacidade destes de analisar uma base de dados de forma eficiente, extraindo e relacionando os dados utilizando um menor número de interações. Mas ao mesmo tempo, os modelos matemáticos estão se tornando cada vez mais complexos e consequentemente, exigindo cada vez mais poder computacional para a sua execução.

Para solucionar o problema da exigência cada vez maior de poder computacional, uma das propostas existentes é a utilização de computação paralela que permitem a utilização da GPU para realização de operações que normalmente seriam realizados na CPU. Como uma GPU possui uma arquitetura altamente paralela, contendo um número muito superior de núcleos se comparada a uma CPU, ela é capaz de realizar os cálculos matriciais propostos pelos modelos matemáticos de forma muito mais eficiente que as CPUs, que realizariam esses mesmos cálculos de forma sequencial ou de forma paralela mas sem a mesma eficiência de uma GPU.

Este trabalho apresentará uma implementação paralela utilizando a plataforma CUDA de um modelo matemático chamado de *Dual Scaling* [2], e proposto por Nishisato. Esse modelo matemático é capaz de modelar um espaço multidimensional

através do mapeamento de itens e transações de uma base de dados, que será utilizado para gerar uma contextualização semântica dos dados, onde a distância entre os pontos representa o quão relacionados são os itens na base de dados.

Mas como se trata de um modelo matemático altamente custoso, a criação de um algoritmo implementando este modelo de forma paralela utilizado GPU é bastante interessante. Para tal, será utilizada a biblioteca Cusp[8] para facilitar a implementação da solução.

Após o detalhamento de solução, este trabalho irá comparar o tempo de execução do algoritmo paralelo proposto com uma solução baseada em CPU desenvolvida utilizando a biblioteca Eigen.

Este trabalho tem por objetivo apresentar uma implementação paralela do modelo matemático chamado de *Dual Scaling*, demonstrar a sua eficiência através de testes comparando-a outras implementações e disponibilizá-la para a comunidade acadêmica.

O código fonte da solução está disponível no repositório para consulta. https://github.com/altobellibm/CEDERJ_2019_LEONARDO_SANTANA_VIEIRA.

2 TRABALHOS RELACIONADOS

Como a quantidade de dados que são produzidos está sempre crescendo, isto torna necessária a evolução proporcional do poder computacional para permitir que o desempenho do processamento dos dados ocorra de maneira satisfatória, como isso não é sempre possível, torna latente a necessidade de otimização das técnicas utilizadas para efetuar o processamento.

Esses problemas de desempenho são visíveis quando a implementação baseada em CPU do *Dual Scaling* é utilizada para processar uma base de dados grande, isso ocorre devido à complexidade do algoritmo que demanda processamento pesado para alcançar o resultado. Neste caso o desempenho não é o ideal, por isso é necessário o desenvolvimento de novas técnicas ou a evolução das técnicas já existente para que seja possível atender a essa demanda.

Como exemplo de implementação baseada em CPU do *Dual Scaling* para bases de dados classificatórios, cito a *Rankcluster*, pacote disponível para a linguagem R disponível no *The Comprehensive R Archive Network*(CRAN) [1]. A linguagem R é um ambiente livre de desenvolvimento de software voltado a aplicações estatísticas.

Este pacote fornece algumas funções importantes, as mais importantes serão descritas abaixo:

- rankclust(): A função principal, responsável pela análise das bases de dados. Esta função só possui um argumento obrigatório, o data, que por sua vez é uma matriz de n transações de classificações ordenadas. A função retorna um objeto da classe ResultTab.
- convertRank(): Converte bases de dados.
- frequence(): Transforma um conjunto de dados brutos em uma matriz de frequência.
- unfrequence(): O oposto de frequence().

Além do Rankcluster, existem outros pacotes disponíveis para R que atacam o mesmo problema como o pmr e o RMallow, mas o Rankcluster é o mais completo pacote disponível no momento.

2.1 DISCUSSÃO

A solução citada sofre de um problema, ela apresenta problemas de desempenho quando é utilizada para processar bases de dados muito grandes. Isso ocorre devido ao fato de ser uma implementação sequencial baseada em CPU, portanto todas as operações realizadas são processadas na CPU, uma por uma de forma sequencial.

O objetivo deste trabalho é apresentar uma implementação da técnica de *Dual Scaling* para bases de dados classificatórios que tire proveito de paralelismo para aumentar o seu desempenho. E a melhor forma de paralelizar o processamento de matrizes é utilizando os recursos disponibilizados por uma GPU já que a sua arquitetura altamente paralela permite ganhos consideráveis de desempenho.

3 FUNDAMENTAÇÃO TEÓRICA

A adaptação de métodos estatísticos clássicos para atender as particularidades de pesquisas científicas, como nas ciências sociais e comportamentais, é algo que vem ocorrendo a várias décadas. Pesquisas nessas áreas se caracterizam por muitas vezes não trabalharem com dados numéricos e sim com escalas de medidas incertas.

Esses tipos de dados representam um desafio porque as relações entre os dados não são exatas, dificultando a sua compreensão.

Para a computação, dados são expressões gerais que descrevem atributos ou características de uma entidade. Dados que variam de uma entidade para outra ou que variam a longo do tempo para uma mesma entidade são chamados de variáveis, ou seja, variáveis representam atributos ou características de uma entidade que são passíveis de serem medidas e podem assumir diversos valores como a cor dos olhos, idade, gênero e tipo sanguíneo.

A análise de dados tem por objetivo extrair tantas informações quanto for possível de dados brutos, utilizando para tal técnicas lógicas e estatísticas para avaliar esses dados.

O *Dual Scaling* é um conjunto de técnicas para análise de dados categóricos. Neste capítulo, será apresentada uma descrição dos tipos de dados que serão analisados neste trabalho e do funcionamento das técnicas que compõem o *Dual Scaling*, além disso, serão apresentados os cálculos realizados pela técnica.

3.1 BASES DE DADOS CATEGÓRICOS

Variáveis categóricas representam os atributos ou características de uma unidade sendo observada, ou seja, uma variável categórica identifica um atributo ou característica de uma unidade sendo observada. Por exemplo, faixa de renda e grau de escolaridade.

As variáveis categóricas podem ser classificadas como dicotômicas, nominais e ordinais.

- Dicotômicas: Variáveis com apenas duas possibilidades de resposta, por isso também chamadas de binárias. Exemplo: Gênero.
- Nominais: Variáveis que identificam um atributo ou característica sem qualquer propriedade em especial. Exemplo: Tipo sanguíneo.
- Ordinais: Variáveis que identificam um atributo ou característica cuja unidade de observação possui propriedades estruturantes, como por exemplo uma ordenação natural. Exemplo: Faixa de renda.

De acordo com Nishisato [2], criador do método, dados categóricos podem ser divididos entre 2 grupos, dados de incidência e dados de dominância.

- Dados de incidência: No grupo de dados de incidência, o *Dual Scaling* tem sido utilizado para os seguintes tipos de dados: Tabelas de frequência e dados de múltipla escolha.
- Dados de dominância: No grupo de dados de dominância, o *Dual Scaling* tem sido utilizado para os seguintes tipos de dados: Dados de ordem de classificação e de comparação pareada.

Este trabalho irá focar no grupo de dados de dominância, mais especificamente, irá utilizar o método para avaliar dados de ordem de classificação.

Dados de ordem de classificação são populares em pesquisas psicológicas e são utilizados para identificar as preferências de uma pessoa. Como exemplo, temos uma pesquisa para identificar as preferências de uma pessoa em relação a cores. Nesta pesquisa o consultado preenche um questionário informando ordenadamente a sua preferência entre as cores pesquisadas. Como pode ser visto na tabela abaixo, as linhas representam as pessoas que responderam o questionário enquanto as colunas representam as cores.

Tabela 1 - Ordem de classificação

Pessoas	Cores					
	Amarelo	Azul	Laranja	Roxo	Verde	Vermelho
Pessoa 1	1	2	3	4	5	6
Pessoa 2	6	5	4	3	2	1
Pessoa 3	2	4	6	5	3	1
Pessoa 4	1	3	5	6	4	2
Pessoa 5	6	4	2	1	3	5
Pessoa 6	5	3	1	2	4	6

3.2 DUAL SCALING

Dual Scaling é um conjunto de técnicas relacionadas para a análise dos mais diversos tipos de dados categóricos ou quantitativos categorizados, incluindo tabelas de contingência, dados de múltipla escolha, dados de ordem de classificação e dados de comparação pareada.

O *Dual Scaling* foi criado para ser aplicado em tabelas baseadas no modelo de Guttman[10, 11], em linhas representam os indivíduos e colunas representam os estímulos. Esses dados são adquiridos através de pesquisas a indivíduos de certos grupos, tendo suas preferências armazenadas e depois mapeadas pelo *Dual Scaling*.

Através do mapeamento, os indivíduos (representados por linhas) e estímulos (representados por colunas) contidos na tabela sob análise são representados como pontos no espaço-solução resultante. Através da visualização da distribuição dos pontos ao longo dos eixos do espaço-solução, pode-se observar como características comuns dentro de um determinado grupo de indivíduos são percebidos pela proximidade dos pontos no espaço-solução. Ao invés disso, características sem denominador comum ficam distantes no espaço-solução.

Ao analisar as respostas dadas por indivíduos a estímulos, o *Dual Scaling* realiza além da análise escalar dos estímulos, a análise quanto as diferenças individuais de escala. Assim, o *Dual Scaling* além de lidar com o problema de calcular o

tamanho das características, também lida com um problema mais amplo que é a classificação de dados.

Os dados categóricos são bastante comuns nas ciências sociais e comportamentais. Neste trabalho, iremos nos aprofundar nos dados de ordem de classificação e apresentar uma implementação de uma técnica de *Dual Scaling* para tratar este tipo de dado.

3.3 ALGORITMO DUAL SCALING EM DADOS DE ORDEM DE CLASSIFICAÇÃO

Neste algoritmo de *Dual Scaling*, recebemos como parâmetro de entrada uma base com dados de ordem de classificação representados em uma matriz de resposta-padrão $A_{n,m}$ de tamanho n x m, onde n é o número de transações (linhas da matriz) que representam cada indivíduo e m é o número de estímulos (colunas da matriz) que representam as opções a serem classificadas. A matriz é preenchida com valores entre 1 e m, onde cada linha representa a classificação daquele indivíduo para os estímulos em consulta. Vale ressaltar que este trabalho não tem o objetivo de explicar a intuição do modelo matemático proposto por Nishisato, vamos apenas apresentar as manipulações matemáticas inerentes do processo do *Dual Scaling*, para mais detalhe do algoritmo veja o livro [12].

Após a definição da base de dados de entrada, a matriz de resposta-padrão é multiplicada pelo número -2, conforme equação abaixo:

$$A = A \times -2 \tag{1}$$

Em seguida, a matriz A é adicionada a uma matriz $B_{n,m}$ cujos elementos são todos iguais a m + 1. A equação pode ser vista abaixo:

$$A = A + B \tag{2}$$

O próximo passo do algoritmo é definir a matriz transposta de A, identificada por C, conforme equação abaixo:

$$C = A^t \tag{3}$$

Em seguida, a matriz A é multiplicada por sua transposta, definida anteriormente como C. A equação pode ser vista abaixo:

$$A = A \times C \tag{4}$$

O próximo passo do algoritmo é multiplicar a matriz A pelo número y definido por $(m \times n \times (m-1) \times (m-1))^{-1}$, conforme equação abaixo:

$$y = \left(m \times n \times (m-1) \times (m-1)\right)^{-1} \tag{5}$$

$$A = A \times y \tag{6}$$

Em seguida, a matriz A é decomposta através da transformação linear, resultando em um vetor de autovalores e uma matriz de autovetores, chamados respectivamente de U e S. Os valores do vetor S devem ser colocados em ordem crescente.

O próximo passo do algoritmo é multiplicar elemento a elemento do vetor S por ele mesmo. Esta operação é chamada de produto de Hadamard [3], e sua equação está abaixo:

$$rho = S \circ S \tag{7}$$

Em seguida, criamos a matriz x e atribuímos a ela a matriz U com exceção da sua coluna mais à direita que é removida.

O próximo passo do algoritmo é multiplicar elemento a elemento da matriz x por ela mesmo. A equação segue abaixo:

$$x_{sqr} = x \circ x \tag{8}$$

Em seguida, criamos o número *ft* que recebe o resultado da operação descrita abaixo:

Observação: n = número de linhas da matriz inicial A e m = número de colunas da matriz inicial A.

$$ft = n \times m \times (m - 1) \quad (9)$$

O próximo passo do algoritmo é multiplicar a matriz x_{sqr} pelo número definido por $n \times (m - 1)$, conforme equação abaixo:

$$z = n \times (m - 1) \quad (10)$$

$$T = x_{sqr} \times z \quad (11)$$

Em seguida, a fim de chegarmos à matriz de pesos padrão dos itens, é preciso antes encontrar o vetor cujos valores representam os multiplicadores das colunas da matriz final de autovetores S. O vetor Cc_{dim} é definido pela equação:

$$Cc_i = \sqrt{\frac{Ft}{Tc_i}} \quad (12)$$

Definido o vetor Cc, precisamos calcular $Nx_{m,dim}$, ou seja, a matriz de pesos padrão dos itens, conforme equação abaixo:

$$Nx_{i,p=S_{i,p}Cc_p} \quad (13)$$

Em seguida, é preciso definir p, que é um vetor que representa os multiplicadores das colunas da matriz de pesos Nx, conforme equação abaixo:

$$p_i = \sqrt{\lambda f_i} \quad (14)$$

E por fim, é preciso encontrar a matriz de pesos projetados dos itens Px que definem as coordenadas dos itens do espaço-solução, conforme a equação abaixo:

$$Px_{i,p} = Nx_{i,p}p_p \quad (15)$$

4 IMPLEMENTAÇÃO PARALELA DO DUAL SCALING UTILIZANDO GPU

O *Dual Scaling* tem o objetivo de analisar e reorganizar os dados de forma a extrair informações destes dados. Mas como os cálculos matemáticos feitos pelo *Dual Scaling* são bastante complexos, isso faz com que as técnicas exijam um alto poder de processamento para serem utilizadas.

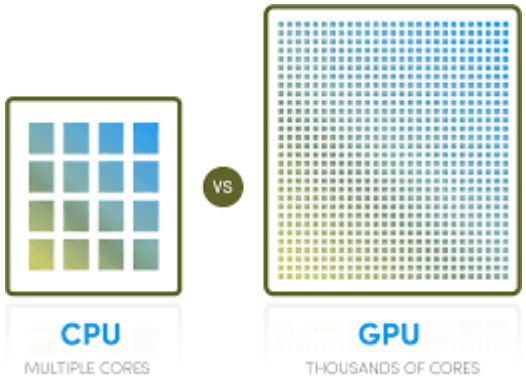
Neste capítulo será descrita uma implementação do *Dual Scaling* paralela afim de minimizar os problemas de desempenho da implementação tradicional do mesmo. Alcançar esse objetivo se torna possível através da utilização da plataforma CUDA, que permite que os cálculos matriciais sejam realizados paralelamente na GPU.

Primeiramente será oferecida uma breve descrição da plataforma CUDA e das bibliotecas utilizadas no desenvolvimento da solução. Após essa descrição, será apresentado o código da solução.

4.1 GPU

GPU [4] é o nome dado a um microprocessador especializado em processar gráficos. Devido a sua arquitetura altamente paralela, GPUs são mais capazes de manipular gráficos computadorizados do que as CPUs.

Figura 1 - CPU vs GPU



4.2 GPGPU

GPGPU[4] é o nome dado ao uso das GPUs em tarefas de natureza não gráfica. Como as APIs gráficas são complexas de serem utilizadas para implementação de algoritmos, se tornou necessário o desenvolvimento de linguagens de alto nível que facilitassem esse processo, permitindo aos desenvolvedores que pudessem abstrair os detalhes de implementação de nível mais baixo das APIs gráficas.

Uma das soluções para este problema foi apresentada pela NVIDIA com a introdução do CUDA.

4.3 CUDA

CUDA [5] é uma plataforma de computação paralela desenvolvida pela NVIDIA que permite que GPUs compatíveis sejam utilizadas para processamento de propósito geral, ou seja GPGPU.

A plataforma CUDA foi desenvolvida de forma a trabalhar com diversas linguagens de alto nível, como C, C++ e Fortran. As aplicações escritas dessa forma executam as suas partes sequenciais na CPU e através de chamadas específicas,

executam as partes paralelizáveis na GPU, poupando a CPU desta carga de trabalho e acelerando a execução da aplicação devido ao nível de paralelização da GPU.

Esse modelo de programação é chamado de computação heterogênea, pois o processamento pode ser realizado por ambos os dispositivos independentemente. No caso do CUDA, a CPU também denominada host é onde é executada a aplicação e é responsável por controlar a gestão de dados, realizar a transferência de memória e configurar a execução da GPU. A GPU também denominada device, é responsável pelo processamento paralelo do algoritmo.

Para a utilização deste modelo, faz-se necessário dividir o problema maior em outros menores, isto requer o ajuste dos algoritmos sequenciais existentes de forma que possam fazer uso do processamento paralelo de forma eficiente.

Para a utilização do CUDA, é preciso uma GPU NVIDIA compatível com o CUDA, além da instalação do SDK da plataforma, chamado CUDA Toolkit, que acompanha o driver de vídeo, o compilador NVCC, além de diversas bibliotecas integradas de forma a facilitar o desenvolvimento.

4.4 THRUST

O Thrust [6] é uma biblioteca de algoritmos paralelos semelhante a Biblioteca Padrão de Modelos do C++ (STL). A interface de alto nível do Thrust permite ao programador grandes aumentos de produtividade ao desenvolver aplicações de alta performance. A biblioteca oferece funções que permitem a simples alocação de memória entre a CPU e a GPU, além de oferecer uma série de funções paralelas, permitindo assim a implementação de algoritmos complexos com um código simples e de fácil compreensão e manutenção.

4.5 CUBLAS

A biblioteca cuBLAS [7] é uma rápida implementação acelerada por GPU para o conjunto de rotinas BLAS (Basic Linear Algebra Subprograms). A biblioteca oferece funções que permitem realizar de forma paralela e eficiente operações de álgebra linear.

4.6 CUSP

O Cusp [8], baseada na Thrust, é uma biblioteca de álgebra linear esparsa e computação gráfica semelhante a Biblioteca Padrão de Modelos do C++ (STL). O Cusp fornece uma interface flexível e de alto nível para manipular matrizes esparsas e resolver sistemas lineares esparsos. A biblioteca oferece funções que permitem ainda mais a simplificação do código, produzindo aplicações de alto desempenho com um código simples e de fácil compreensão e manutenção.

4.7 CODIFICAÇÃO EM CUDA

Na codificação do algoritmo de *Dual Scaling* para dados de ordem de classificação foi utilizada a linguagem de programação C++ empregando uma implementação paralelizada em GPU utilizando a arquitetura CUDA e com auxílio das bibliotecas Cusp, Thrust e cuBLAS para a que a codificação fique em mais alto nível, facilitando a implementação do código e seu entendimento.

Seguem abaixo as definições dos functores que serão utilizados ao longo do código, a sua utilidade será explicada quando os mesmos forem utilizados.

```
template <typename T>
struct linear_index_to_row_index : public thrust::unary_function<T,T>
```

```
{
  T C;

  __host__ __device__
  linear_index_to_row_index(T C) : C(C) {}

  __host__ __device__
  T operator()(T i)
  {
    return i / C;
  }
};

template <typename T>
    struct reciprocal_my : public thrust::unary_function<T,T>
    {
      T value;
      reciprocal_my(T thr) : value(thr) {};
      __host__ __device__
      T operator()(const T& v) const {
          return sqrt(T(value) / v);
      }
    };

template<typename T>
struct is_true: thrust::unary_function<T, T> {
    T col;

    is_true(T _c) :
            col(_c) {
    }
    ;

    __host__ __device__
    bool operator()(const T &x)
    {
        return (x % col) != 0;
    }
};

template<typename T>
struct sub_matrix: thrust::unary_function<T, T> {
    T col;
    T row;
    T pitch;

    sub_matrix(T _c, T _r, T _p) :
            col(_c), row(_r), pitch(_p) {
    };

    __host__ __device__
    bool operator()(const T &x)
    {
        return  (x % (int)pitch) < (int) col && (x / (int)pitch) <
(int)row;
    }
};

template <typename T>
    struct column_by_vector : public thrust::unary_function<T,T>
```

```
    {
      T* data;
      T col;
      column_by_vector(T *_data, T _col) : data(_data), col(_col) {};
      __host__ __device__
      T operator()(const thrust::tuple<int,type>& v) {
          return data[thrust::get<0>(v) % (int)col] * thrust::get<1>(v);
     }
    };
```

O algoritmo começa recebendo um arquivo contendo uma base da dados de ordem de classificação. Este arquivo é então lido e os valores são copiados para um vetor chamado database, além disso durante a leitura também é calculado o número de linhas e colunas da base de dados, conforme abaixo:

```
    if (argc != 2) {
        cout << "Argumento incorreto. " << endl;
        cout << "parâmetro entrada: ./main <base de dados> " << endl;
        return 1;
    }

    std::string strFile = argv[1];
    std::string strPrefixFile = strFile.substr(str-
File.find_last_of("/")+1,strFile.find(".dat")-4);

    ifstream file;
    file.open(strFile);

    if(!file.is_open()){
        std::cout << "Error opening file" << std::endl;
        return 1;
    }

    int intNrLinhas = 0;
    int intNrColunas = 0;
    std::string strLinha;
    std::string strNumero;
    std::vector<type> database;

    while (getline(file, strLinha)) {
        stringstream ssLinha(strLinha);
        while (getline(ssLinha, strNumero, ' ')) {
            database.push_back(type(std::stoi(strNumero)));
            if ( intNrLinhas == 0 ){
                intNrColunas++;
            }
        }
        intNrLinhas++;
    }

    file.close();
```

Após a leitura do arquivo, é criado um objeto chamado E do tipo Array2D da biblioteca Cusp que representa uma matriz densa cujas dimensões são definidas pelos números de linhas e colunas calculados anteriormente.

```
Array2d E(intNrLinhas, intNrColunas);
thrust::copy(database.begin(), database.end(), E.values.begin());
```

Após a criação da Matriz E, são realizadas as seguintes operações:

1. A matriz E é multiplicada por -2.[Equação (1)]
2. Aos elementos da matriz E é somado um valor igual ao número de colunas +1.[Equação (2)]
3. É criada a matriz matrizTransposta que é a matriz transposta de E.[Equação (3)]

```
thrust::transform(E.values.begin(), E.values.end(),
        E.values.begin(), cusp::multiplies_value<int>(-2));

thrust::transform(E.values.begin(), E.values.end(),
        E.values.begin(), cusp::plus_value<int>(E.num_cols + 1));

Array2d matrizTransposta;

cusp::transpose(E, matrizTransposta);
```

Agora é criada a matriz C com ambas as dimensões iguais ao número de colunas de E e que recebe o resultado da multiplicação de matrizes entre E e matrizTransposta. Para esse cálculo é utilizada a função cublaSgemm oferecida pela biblioteca cuBLAS.[Equação (4)]

Embora exista uma função multiply disponível na biblioteca Cusp para multiplicação de matrizes, como essa função estava gerando um alto custo de tempo de execução, foram analisadas outras possibilidades e a solução foi utilizar a função cublaSgemm da biblioteca cuBLAS para realizar essa multiplicação.

```
Array2d C(E.num_cols, E.num_cols);

cublasHandle_t h;
cublasCreate(&h);
float alpha = 1.0f;
float beta = 0.0f;
```

```
    cublasSgemm(h, CUBLAS_OP_N, CUBLAS_OP_N, E.num_cols, matrizTrans-
posta.num_rows, matrizTransposta.num_cols, &alpha, thrust::raw_poin-
ter_cast(E.values.data()), E.num_cols, thrust::raw_pointer_cast(ma-
trizTransposta.values.data()), matrizTransposta.num_cols, &beta,
thrust::raw_pointer_cast(C.values.data()), E.num_cols);
```

Depois do preenchimento da matriz C, cada elemento desta matriz é divido pelo valor definido abaixo:[Equações (5, 6)]

```
    thrust::transform(C.values.begin(), C.values.end(), C.values.begin(),
        cusp::divide_value<type>(
        E.num_cols * E.num_rows * (E.num_cols - 1)
            * (E.num_cols - 1)));
```

O próximo passo do algoritmo é calcular o vetor de autovalores S e a matriz de autovetores U, para tal foi utilizada a função svd, escrita sem a utilização de bibliotecas intermediárias de alto nível e por isso o seu código é mais complexo. Nesta implementação foi utilizada alocação de ponteiros com device_ptr da biblioteca Thrust além da função cudaMalloc da biblioteca padrão do CUDA.

```
    cusp::array2d<type, cusp::device_memory> U(C.num_cols, C.num_cols);
    cusp::array1d<type, cusp::device_memory> S(C.num_cols);

    svd(C.num_cols, C, U, S);


bool svd(int M, cusp::array2d<type, cusp::device_memory>& M_denseHM,
        cusp::array2d<type, cusp::device_memory>& U,
        cusp::array1d<type, cusp::device_memory>& S) {

    thrust::device_ptr<type> dev_ptr = &(M_denseHM.values[0]);
    type *M_raw_ptr = thrust::raw_pointer_cast(dev_ptr);

    int work_size = 0;

    int *devInfo;
    cudaMalloc(&devInfo, sizeof(int));
    type *d_U;
    cudaMalloc(&d_U, M * M * sizeof(type));
    type *d_V;
    cudaMalloc(&d_V, M * M * sizeof(type));
    type *d_S;
    cudaMalloc(&d_S, M * sizeof(type));

    cusolverDnHandle_t solver_handle;
    cusolverDnCreate(&solver_handle);

    cusolverDnSgesvd_bufferSize(solver_handle, M, M, &work_size);

    type *work;
```

```
    cudaMalloc(&work, work_size * sizeof(type));

    cusolverDnSgesvd(solver_handle, 'A', 'A', M, M, M_raw_ptr, M, d_S, d_U,
M,
            d_V, M, work, work_size, NULL, devInfo);

    int devInfo_h = 0;
    cudaMemcpy(&devInfo_h, devInfo, sizeof(int), cudaMemcpyDeviceToHost);

    thrust::device_ptr<type> dev_ptr_U(d_U);
    thrust::copy(thrust::device, dev_ptr_U, dev_ptr_U + (M * M),
            U.values.begin());

    thrust::device_ptr<type> dev_ptr_S(d_S);
    thrust::copy(thrust::device, dev_ptr_S, dev_ptr_S + M, S.begin());

    cusolverDnDestroy(solver_handle);

    return 1;
}
```

Após a decomposição, é criado um objeto chamado rho do tipo Array1d da biblioteca Cusp que representa um vetor. O vetor rho recebe como valores o quadrado de cada elemento do vetor de autovalores S exceto a última coluna.[Equação (7)]

```
    Array1d rho(E.num_cols - 1);

    thrust::transform(S.begin(), S.end()-1, rho.begin(),
            cusp::sqrt_functor<type>());
```

A seguir é criada a matriz x_sqr que recebe como valores o quadrado de cada elemento da matriz de autovetores U exceto a última coluna.[Equação (8)]

```
    Array2d x(E.num_cols, E.num_cols - 1);

    cusp::array1d<int,cusp::device_memory> index(intNrColunas*intNrColu-
nas);
    thrust::sequence(index.begin(), index.end(), 1);

    thrust::copy_if(U.values.begin(), U.values.end(), index.begin(), x.va-
lues.begin(), is_true<int>(intNrColunas));


    Array2d x_sqr(x.num_rows, x.num_cols);

    thrust::transform(x.values.begin(), x.values.end(), x_sqr.values.be-
gin(),
            cusp::square_functor<type>());
```

Abaixo é calculado o valor de ft que será usado mais tarde.[Equação (9)]

```
    int ft = intNrLinhas * intNrColunas * (intNrColunas - 1);
```

No próximo passo é calculada a matriz T, cujos valores são o resultado da multiplicação da matriz x_sqr pelo valor definido abaixo.[Equações (10, 11)]

```
    Array2d T = x_sqr;
    thrust::transform(T.values.begin(), T.values.end(),
            T.values.begin(), cusp::multiplies_value<type>(intNrLinhas *
(intNrColunas - 1)));
```

Após o cálculo da matriz T, é calculada a matriz x_normed, para este cáculo é utilizado o valor de ft calculado anteriormente. Primeiro é definida a transposta de T para facilitar o cálculo, depois é utilizada a função thrust::tabulate e o functor linear_index_to_row_index para obter o vetor de frequências das colunas de T, após isto é feito um somatório dos valores de cada índice repetido da matriz utilizando a função thrust::reduce_by_key, depois é utilizada a função thrust::transform junto ao functor reciprocal_my para realizar o cálculo do vetor Cc e por fim é calculada a matrix x_normed, para isso é utilizada a função thrust::transform e o functor column_by_vector para multiplicar o vetor Cc pela matriz de autovetores x.[Equações 12, 13, 14]

```
    cusp::array1d<int,cusp::device_memory> index_sum_t(intNrColunas);
    cusp::array1d<type,cusp::device_memory> marginal_sum_t(intNrColunas-1);

    cusp::array2d<type,cusp::device_memory> T_T(intNrColunas,intNrLinhas);
    cusp::transpose(T, T_T);

    Array1d index_similar(intNrColunas*(intNrColunas - 1));
    thrust::tabulate(thrust::device, index_similar.begin(), index_simi-
lar.end(), linear_index_to_row_index<int>(intNrColunas));

    thrust::reduce_by_key(index_similar.begin(),
                          index_similar.end(),
                          T_T.values.begin(),
                          index_sum_t.begin(),
                          marginal_sum_t.begin(),
                          thrust::equal_to<int>(),
                          thrust::plus<type>());

    cusp::array1d<type,cusp::device_memory> cc(intNrColunas-1);

    thrust::transform(marginal_sum_t.begin(), marginal_sum_t.end(), cc.be-
gin(), reciprocal_my<type>(type(ft)));

    Array1d index_X(intNrColunas*(intNrColunas - 1));
    thrust::sequence(index_X.begin(), index_X.end(),0);
```

```
    cusp::array2d<type,cusp::device_memory> x_nor-
med(x.num_rows,x.num_cols);

    thrust::transform(thrust::make_zip_iterator(thrust::make_tuple(in-
dex_X.begin(), x.values.begin())), thrust::make_zip_itera-
tor(thrust::make_tuple(index_X.end(), x.values.end())), x_normed.values.be-
gin(), column_by_vector<type>(thrust::raw_poin-
ter_cast(cc.data()),(type)x.num_cols));
```

E por último é calculada a matrix x_project utilizando a matriz x_normed calculada anteriormente. Para tal é utilizada a função thrust::transform e o functor column_by_vector para realizar este cálculo.[Equação (15)]

```
    cusp::array2d<type,cusp::device_memory> x_pro-
ject(x.num_rows,x.num_cols);

    thrust::transform(thrust::make_zip_iterator(thrust::make_tuple(in-
dex_X.begin(), x_normed.values.begin())), thrust::make_zip_itera-
tor(thrust::make_tuple(index_X.end(), x_normed.values.end())), x_pro-
ject.values.begin(), column_by_vector<type>(thrust::raw_poin-
ter_cast(rho.data()),(type)x.num_cols));
```

5 TESTES E RESULTADOS

Este capítulo apresenta os resultados obtidos nos testes realizados para a avaliação da implementação do algoritmo paralelizado utilizando a GPU, comparando-a com outra implementação sequencial utilizando CPU.

As bases de dados utilizadas nos testes foram criadas especificamente com este propósito, de forma a acentuar o impacto da variação dos números de linhas e colunas na performance das implementações. As bases de dados possuem um número n de linhas e m de colunas, o algoritmo de criação recebe esses valores e então cria a base com essas dimensões e preenche cada linha com números aleatórios entre 1 e m sem repetir o mesmo número, dessa forma simulando que cada transação represente uma classificação de um usuário.

Os testes foram realizados com o objeto de avaliar o tempo de execução dos algoritmos e as bases de dados foram divididas em 2 grupos, o primeiro tem estabelecido o limite de 500 e o segundo tem estabelecido o limite de 5000, as bases foram criadas conforme abaixo:

Tabela 2 - Bases de dados entre 100 e 500

Bases de Dados	Linhas	Colunas
Base de Dados 1	100	500
Base de Dados 2	200	500
Base de Dados 3	300	500
Base de Dados 4	400	500
Base de Dados 5	500	500
Base de Dados 6	500	100
Base de Dados 7	500	200
Base de Dados 8	500	300
Base de Dados 9	500	400

Tabela 3 - Bases de dados entre 1000 e 5000

Bases de Dados	Linhas	Colunas
Base de Dados 1	1000	5000
Base de Dados 2	2000	5000
Base de Dados 3	3000	5000
Base de Dados 4	4000	5000
Base de Dados 5	5000	5000
Base de Dados 6	5000	1000
Base de Dados 7	5000	2000
Base de Dados 8	5000	3000
Base de Dados 9	5000	4000

As especificações do computador e da GPU utilizados seguem abaixo:

Tabela 4 - Especificação do computador

Processador	AMD Ryzen 5 2600X
Memória RAM	Crucial 16 GB(8GB x 2) DDR4 3000 MHz CL15
Sistema Operacional	Arch Linux (Linux 5.3.13.1)
Placa de Vídeo	Asus Phoenix NVIDIA GeForce GTX 1050 TI
Versão do CUDA	10.1.243
Versão do Driver	440.36

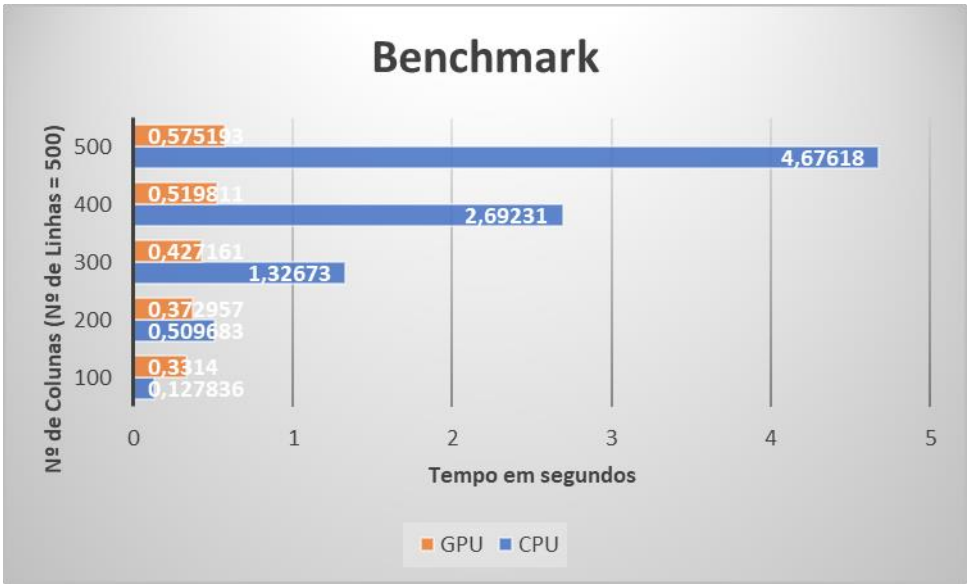
Tabela 5 - Especificação da GPU

Arquitetura da GPU	Pascal
CUDA Compute Capability	6.1
NVIDIA CUDA Cores	768
Clock da GPU padrão	1290 MHz
Clock da GPU aumentado	1392 MHz
Memória de Vídeo	GDDR5 4GB
Clock de memória	7008 MHz
Interface de memória	128-bits

A primeira implementação testada é a proposta neste trabalho, foi escrita na linguagem C++ para a plataforma CUDA, utilizando as bibliotecas Cusp, Thrust e cuBLAS para auxiliar o desenvolvimento do algoritmo paralelo. Já a segunda implementação[13] foi escrita também em C++ mas de forma sequencial a ser executada

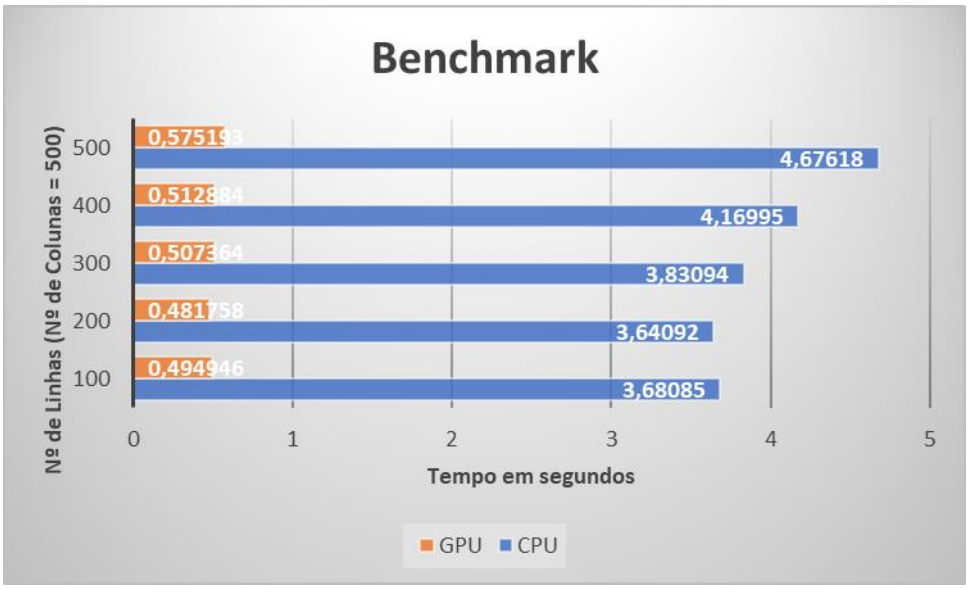
na CPU utilizando a biblioteca Eigen [9]. A primeira implementação será denominada nos gráficos como GPU e a segunda como CPU. Não fizemos a comparação com a biblioteca em R, que também é sequencial por sem em outra linguagem.

Gráfico 2 - Valor fixo = 500 Linhas



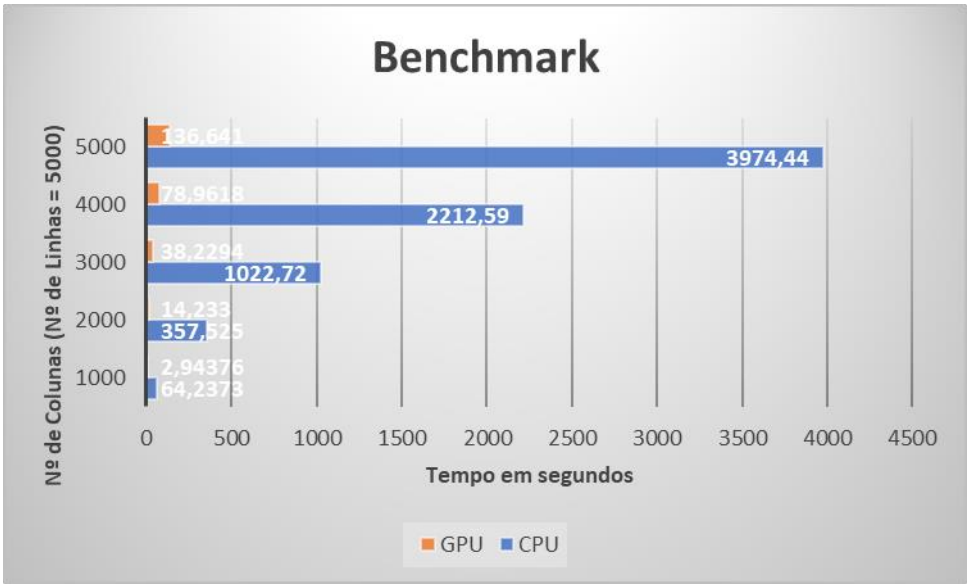
No gráfico 2 avaliamos o impacto da variação do número de colunas para as 2 implementações. Neste caso é estabelecido um valor fixo para o número de linhas, no caso 500, enquanto o número de colunas varia de 100 em 100 até 500.

Gráfico 3 - Valor fixo = 500 Colunas



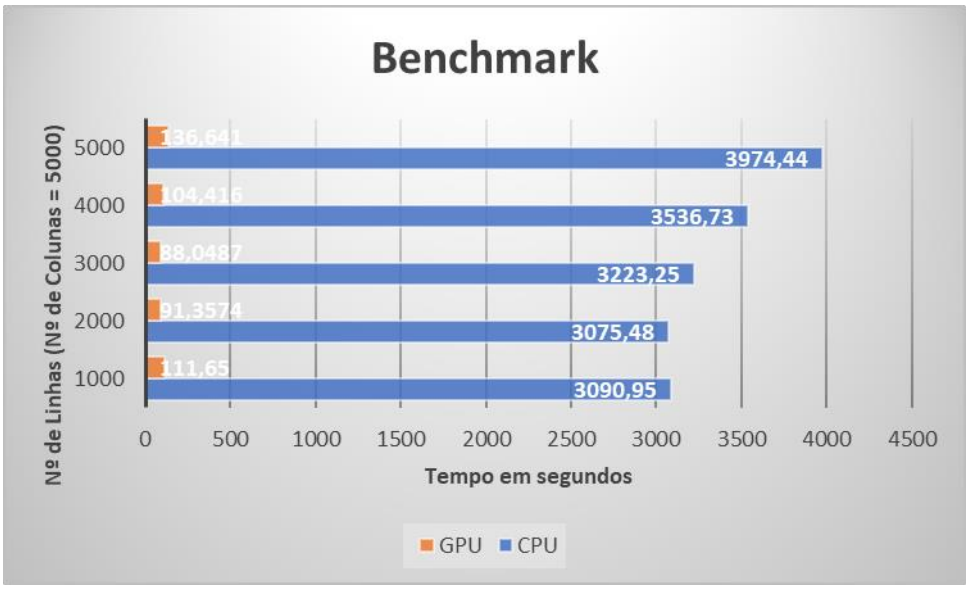
No gráfico 3 avaliamos o impacto da variação do número de linhas para as 2 implementações. Neste caso é estabelecido um valor fixo para o número de colunas, no caso 500, enquanto o número de linhas varia de 100 em 100 até 500.

Gráfico 4 - Valor fixo = 5000 Linhas



No gráfico 4 avaliamos o impacto da variação do número de colunas para as 2 implementações. Neste caso é estabelecido um valor fixo para o número de linhas, no caso 5000, enquanto o número de colunas varia de 1000 em 1000 até 5000.

Gráfico 5 - Valor fixo = 5000 Colunas



No gráfico 5 avaliamos o impacto da variação do número de linhas para as 2 implementações. Neste caso é estabelecido um valor fixo para o número de colunas, no caso 5000, enquanto o número de linhas varia de 1000 em 1000 até 5000.

Como pode ser observado nos gráficos 2, 3, 4 e 5, a implementação em GPU foi mais rápida em quase todos os testes, chegando a ser aproximadamente 36,5x mais rápida no teste com base de 3000 linhas e 5000 colunas como pode ser visto no gráfico 5. O único teste na qual não foi mais rápida foi no teste com base de 500 linhas e 100 colunas, como pode se ver no gráfico 5, a implementação em CPU foi aproximadamente 2,5x mais rápida, isto ocorreu devido ao fato de que em bases bem pequenas como esta, o tempo gasto transferindo a base de dados da memória principal para a memória da GPU mais o tempo de inicialização da *engine* CUDA acaba sendo proporcionalmente muito grande se comparado ao tempo de execução do algoritmo, o que faz com que a implementação em GPU não compense nesses casos.

Outra informação relevante que pode ser obtida analisando os gráficos é o impacto do aumento do número de colunas(gráficos 2 e 4) se comparado ao aumento do número de linhas(gráficos 3 e 5). Enquanto o aumento no tempo de execução dos dois algoritmos no caso de aumento do número de linhas é linear, no caso de aumento do número de colunas, esse crescimento passa a ser exponencial para a implementação em CPU e são justamente esses casos que demonstraram a maior diferença de performance entre as duas implementações.

6 CONCLUSÕES E TRABALHOS FUTUROS

Neste trabalho foi discutida a necessidade da busca por soluções mais eficientes para a análise de grandes volumas de dados. O cenário atual, de crescimento exponencial da produção de dados, tornou imprescindível a busca por novas ferramentas mais eficientes para atender esta demanda de análise de dados.

Assim, o algoritmo de *Dual Scaling* aparece como uma solução para produzir informação identificando correlações entre itens pertencentes à base de dados. Mas como essa técnica exige um alto custo de processamento, este trabalho propõe uma implementação paralela em GPU utilizando a plataforma CUDA com o objetivo de maximizar a eficiência da técnica em relação ao custo computacional.

Com base nos resultados obtidos pelos testes, podemos observar que a implementação proposta oferece um substancial ganho de desempenho se comparado a implementação sequencial em CPU.

Como trabalho futuro, seria interessante um estudo mais aprofundado sobre a relação entre o tamanho da base de dados e o tempo de transferência desta da memória principal para a memória da GPU. Seria interessante também realizar novos testes desta implementação utilizando diferentes GPUs a fim de verificar a escalabilidade da implementação em relação ao número de núcleos CUDA e a diferentes graus de performance produzidos por diferenças de arquitetura, quantidade e velocidade da memória e a velocidade da GPU.

7 REFERÊNCIAS BIBLIOGRÁFICAS

1. *CRAN*. (s.d.). Fonte: The Comprehensive R Archive Network: https://cran.r-project.org/.
2. Nishisato, S. (December de 1993). On quantifying different types of categorical. *Psychometrika, 58*(4), 617-629.
3. Johnson, C. R. (1974). Hadamard products of matrices. *Linear and Multilinear Algebra, 1*(4), 295-307.
4. Gomes, S. E. (s.d.). Um Tutorial sobre GPUs. Fonte: http://www.cecm.usp.br/~selune/mysite/tutorialGPUs-resumo.html
5. NVIDIA. (s.d.). GPU Accelerated Computing with C and C++. Fonte: https://developer.nvidia.com/how-to-cuda-c-cpp
6. Hoberock, J., & Bell, N. (s.d.). Thrust. Fonte: https://thrust.github.io/
7. NVIDIA. (s.d.). cuBLAS. Fonte: https://developer.nvidia.com/cublas
8. NVIDIA. (s.d.). CUSP. Fonte: https://cusplibrary.github.io/
9. eigen@lists.tuxfamily.org. (s.d.). Eigen is a C++ template library for linear algebra: matrices, vectors, numerical solvers, and related algorithms. Fonte: http://eigen.tuxfamily.org/index.php?title=Main_Page
10. Guttman, L. A. (1944). A basis for scaling qualitative data. American Sociological Review, 91, 139-150.
11. Guttman, L. A. (1950). The basis for scalogram analysis. Em S. A. Stouffer, L. A. Guttman, & E. A. Schuman, Measurement and Prediction: Studies in Social Psychology in World War II. Princeton University Press.
12. Nishisato, S. (1994). Elements of Dual Scaling: An Introduction to Practical Data Analysis. HIllsdale, NJ: Psychology Press.
13. Mantuan, A. d. (2016). CONTEXTUALIZAÇÃO ESPACIAL PARA MINERAÇÃO DE ITEMSETS RAROS OU FREQUENTES NÃO-REDUNDANTES EM BASES DE DADOS. Niterói, RJ: Dissertação de Mestrado (Mestre em Computação), Programa de Pós-Graduação em Computação, UFF.