UNIVERSIDADE FEDERAL DO RIO DE JANEIRO

INSTITUTO DE MATEMÁTICA

CURSO DE BACHARELADO EM CIÊNCIA DA COMPUTAÇÃO

Jorge Rama Krsna Mandoju

Implementação do algoritmo Apriori usando GPU: um estudo de caso

RIO DE JANEIRO

2018

Jorge Rama Krsna Mandoju

Implementação do algoritmo Apriori usando GPU: um estudo de caso

Trabalho de conclusão de curso de graduação apresentado ao Departamento de Ciência da Computação da Universidade Federal do Rio de Janeiro como parte dos requisitos para obtenção do grau de Bacharel em Ciência da Computação. (justificado sem recuo, início no meio da página).

Orientador: Profa. Valeria Menezes Bastos

RIO DE JANEIRO

2018

CIP - Catalogação na Publicação

Mandoju, Jorge Rama Krsna
M271i Implementação do algoritmo Apriori usando GPU: um estudo de caso / Jorge Rama Krsna Mandoju. -- Rio de Janeiro, 2018.
32 f.

Orientadora: Valeria Menezes Bastos.
Trabalho de conclusão de curso (graduação) - Universidade Federal do Rio de Janeiro, Instituto de Matemática, Bacharel em Ciência da Computação, 2018.

1. Programação paralela na GPU. 2. Árvore B+. 3. Apriori. I. Bastos, Valeria Menezes, orient. II. Título.

Elaborado pelo Sistema de Geração Automática da UFRJ com os dados fornecidos pelo(a) autor(a).

Jorge Rama Krsna Mandoju

Implementação do algoritmo Apriori usando GPU: um estudo de caso

Trabalho de conclusão de curso de graduação apresentado ao Departamento de Ciência da Computação da Universidade Federal do Rio de Janeiro como parte dos requisitos para obtenção do grau de Bacharel em Ciência da Computação.

Aprovado em 25 de Abril de 2018.

BANCA EXAMINADORA:

Prof. Valeria Menezes Bastos, D. Sc (UFRJ)

Prof. Adriana Santarosa Vivacqua, D. Sc (UFRJ)

Prof. Myrian Christina de Aragão Costa, D. Sc (UFRJ)

RESUMO

O Apriori é um algoritmo de mineração para classificação de dados. Durante sua execução, realiza várias combinações entre os dados de modo a encontrar as relações mais frequentes. Porém, devido à capacidade de processamento em ambientes computacionais regulares, o seu desempenho é ruim quando é considerado um elevado número de informações para processar. Uma boa maneira de se aproximar da solução do problema com uma velocidade de processamento adequada é utilizando GPUs. Dessa forma, é possível obter um alto grau de paralelização, acelerando a execução do algoritmo. Entretanto, para a implementação do mesmo, o algoritmo foi modificado, para atender ao paradigma de programação CUDA e dessa forma, ser suportado pela arquitetura GPU. Este trabalho tem como objetivo avaliar o desempenho do algoritmo Apriori, utilizando a plataforma de computação paralela e modelo de programação CUDA, em uma placa de vídeo GeForce GTX 980 ti.

Palavras-chave: Apriori. Bitmap. CUDA. Árvore B+. Benchmark.

ABSTRACT

Apriori is a mining algorithm used for data classification. During its execution, it makes several combinations between data in order to find the most frequent sets. However, it shows poor performance for a large number of information due to low capacity of regular computing environments. A good approach with an appropriate processing speed is to use GPUs. Therefore, it is possible to obtain a high degree of parallelization, speeding up algorithm execution. However, the algorithm has been modified for CUDA programming paradigm, and thus, be supported by GPU architecture. This work aims to evaluate the performance of the Apriori algorithm, using a parallel computing platform and CUDA programming model, on a GeForce GTX 980 ti card.

Palavras-chave: Apriori. Bitmap. CUDA. B+ tree. Benchmark.

LISTA DE ILUSTRAÇÕES

LISTA DE TABELAS

LISTA DE SIGLAS

CPU – Central Processing Unit

GPU – Graphical Processing Unit

CUDA – Compute Unified Device Archtecture

SUMÁRIO

1 INTRODUÇÃO

Encontrar os conjuntos de itens frequentes é um dos métodos para mineração de dados (ULLMAN, 2000). Este método ajuda a encontrar quais são as relações mais comuns no conjunto de dados de que está trabalhando.

O Apriori é um algoritmo que faz aplicação deste método procurando os conjuntos mais frequentes, começando por aqueles com apenas um elemento. Em seguida, o algoritmo identifica os conjuntos frequentes com mais elementos relacionados, considerando que caso um conjunto X não é frequente, então sabe-se que um conjunto Y que possui X como subconjunto também não será frequente. Esta consideração faz com que não seja necessário calcular a frequência de todos os conjuntos para descobrir quais são os mais frequentes.

Porém, este algoritmo acaba se tornando bastante custoso pois além de ter que realizar as combinações entre os elementos, é necessário realizar a contagem de cada conjunto. Quando o número de combinações fica muito grande, o algoritmo consome muito tempo de processamento. Uma alternativa para reduzir este tempo é realizar o processamento em GPU.

Para utilizar GPU, foi necessário realizar modificações no funcionamento e estrutura do algoritmo, como proposto por Zhang (2011). Estas modificações servem para que o problema se encaixe na arquitetura CUDA, principalmente para favorecer o conceito de grade e blocos.

Este trabalho irá avaliar o desempenho do algoritmo em um ambiente doméstico, utilizando uma placa de vídeo geforce GTX 980 ti.

1.1 MOTIVAÇÃO

A motivação do trabalho foi verificar o quão a utilização de GPU, para processamento de algoritmos de mineração de dados em um ambiente doméstico, melhora o desempenho dos algoritmos. Apesar de algumas implementações no passado utilizarem GPUs (FANG, 2009), os ambientes em que eram utilizados estes algoritmos eram dedicados à utilização de GPUs com placas de vídeo dedicadas a resolução deste problema como, por exemplo, as placas de vídeo Tesla. Além disto, a biblioteca CUDA foi bastante modificada, para atender aos paradigmas de programação paralela.

O próprio GPapriori, feito por Zhang (2011) era uma implementação que utilizava uma placa de vídeo Nvidia Tesla T10 que possui uma arquitetura Tesla 2.0. Na época em que

este algoritmo foi criado, a biblioteca CUDA utilizada para gerar o trabalho não tinha muitas funcionalidades que possui hoje.

O Bitmap apriori, criado por Sandaruwan (2011), propôs a utilização do Apriori nas GPUs utilizando os índices bitmap.

Na área de aprendizado de máquina, a computação paralela utilizando GPUs é muito utilizada, pois permite maior desempenho dos algoritmos, diante da enorme quantidade de cálculos a serem executados. A linguagem OptiML (SUJEETH, 2011), foi criada com intuito de automatizar o paralelismo dos algoritmos de aprendizado de máquina, gerando automaticamente códigos feito em CUDA.

1.2 OBJETIVO

O principal objetivo deste trabalho é aprender a utilizar o novo paradigma de programação na arquitetura CUDA, dadas as diferenças na arquitetura de um processador X86. Com isto, foram verificados na prática alguns problemas comuns que sempre são mencionados como, por exemplo, o tempo de transferência do barramento de dados entre CPU e GPU.

Para ajudar neste aprendizado, foi implementado o algoritmo GPapriori, criado por Zhang (2011) em uma placa de vídeo com arquitetura Maxwell e foi feita a comparação dos tempos e resultados contra a implementação utilizando somente CPU.

1.3 APRESENTAÇÃO DO DOCUMENTO

Este documento tem no capítulo 2 uma apresentação do conceito teórico utilizado no trabalho, identificando primeiramente as características da arquitetura CUDA e os algoritmos utilizados na execução do trabalho.

O capítulo 3 apresenta a metodologia do trabalho, demonstrando o ambiente em que foi desenvolvido, as bases de dados utilizadas nos testes, a forma de implementação do algoritmo em C++, e a métrica para avaliação dos resultados.

O capítulo 4 descreve todos os resultados obtidos pelas execuções dos testes. O capítulo 5 contém as conclusões a respeito dos resultados, com possíveis trabalhos futuros para serem feitos.

2 CONCEITUAÇÃO

2.1 HARDWARE

2.1.1 Arquitetura Cuda

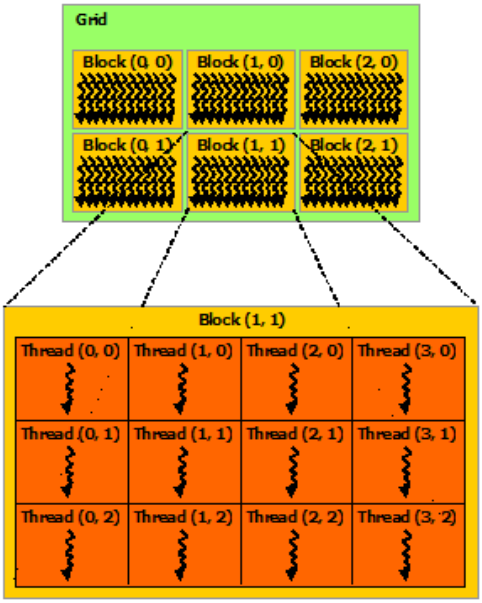
A arquitetura CUDA – Compute Unified Device Archtecture (NVIDIA, 2017) é uma arquitetura criada pela Nvidia, a qual possibilita a execução de algoritmos gerais utilizando a GPU, como por exemplo, algoritmos de aprendizado de máquina, a fim de aproveitar o alto grau de paralelização que as GPUs possuem. Para utilizar esta arquitetura, é necessário desenvolver os programas na linguagem C, criando um nível de abstração nos algoritmos manipulados pela própria GPU, porém, outras interfaces de programação são suportadas, tais como C++, Fortran e Java.

A arquitetura possui dois tipos de processadores: Streaming MultiProcessors (multiprocessadores) e Scalar Processors (núcleos de processamento). Cada multiprocessador possui vários núcleos de processamento. Os multiprocessadores trabalham de forma totalmente independente e paralela entre si, porém eles possuem a arquitetura SIMT (Single Instruction, Multiple Thread), a qual todos os núcleos de um mesmo grupo ou multiprocessador executam a mesma instrução de forma paralela.

2.1.2 Conceito de Grade e blocos

Segundo a Nvidia (2017), CUDA utiliza o conceito de grade e blocos para gerenciar os threads. Uma grade possui vários blocos, onde cada bloco é gerenciado por um multiprocessador e possui um número máximo de threads que podem ser alocadas de acordo com cada grade ou GPU. As grades organizam os blocos em forma de uma matriz bidimensional, e os blocos organizam cada thread de maneira tridimensional, conforme apresentado na Figura 1.

Figura 1: Demonstração da arquitetura de grid e bloco da arquitetura CUDA.

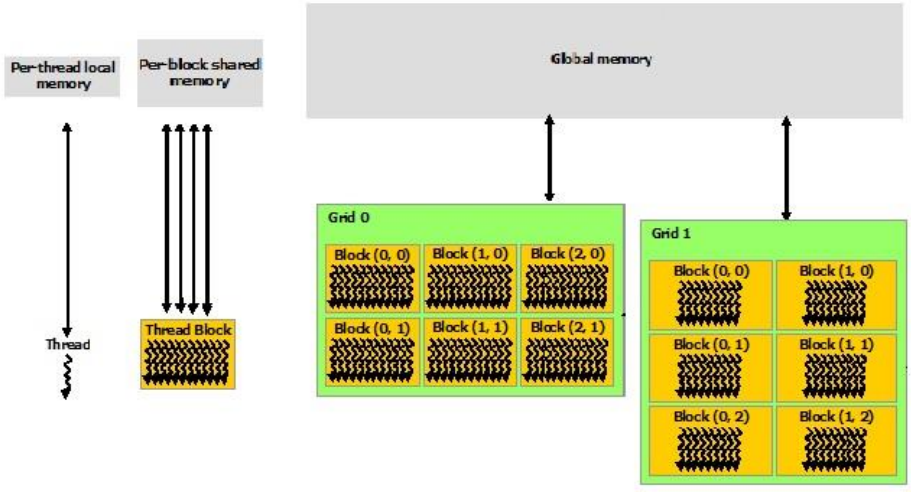


Fonte: Nvidia (2017)

2.1.3 Memória hierárquica da arquitetura CUDA

De acordo com a Nvidia (2017), a arquitetura CUDA possui uma memória hierárquica. Há uma memória global que é acessada por grade, e cada bloco possui uma memória local que é compartilhada somente com os threads do mesmo bloco, e por fim, cada thread possui uma memória local própria. Na figura 2 é possível ver a forma de organização das memórias.

Figura 1: Demonstração da hierarquia da memória na arquitetura CUDA.



Fonte: Nvidia (2017)

2.2 SOFTWARE

2.2.1 Apriori

O algoritmo Apriori criado por Agrawal (1994), é utilizado para fazer a mineração de conjuntos frequentes e aprender regras de associação através de um banco de dados transacional.

Primeiramente, antes de iniciar o algoritmo, é necessário definir o suporte mínimo como parâmetro de entrada. O suporte é o número de transações que um determinado conjunto de dados aparece em relação ao total de transações (SILVA, 2004). Caso o suporte seja muito alto, a porcentagem de conjuntos existentes no banco de dados também deve ser alta, e com isso, o algoritmo apresenta poucos resultados. Caso o suporte seja muito baixo, o algoritmo gera um número alto de associações, e associações pouco relevantes (com um índice de frequência extremamente baixo), influenciando no desempenho do algoritmo, pois conjuntos poucos frequentes serão calculados.

No primeiro passo do algoritmo, o Apriori simplesmente conta o número de ocorrências de cada item, a fim de determinar os maiores conjuntos formados por apenas um elemento.

No segundo passo, o Apriori gera os subconjuntos possíveis de cada conjunto gerado pelo passo anterior. Após obter todos os subconjuntos, é realizado um filtro com base na frequência de cada um desses subconjuntos utilizando o suporte, ou seja, retirando todos os subconjuntos cuja frequência é menor que o suporte. Este passo é conhecido como poda dos candidatos.

Após isto, o algoritmo volta para o passo 2, fazendo o cálculo de todos os subconjuntos novamente, até que não se tenha nenhum subconjunto gerado.

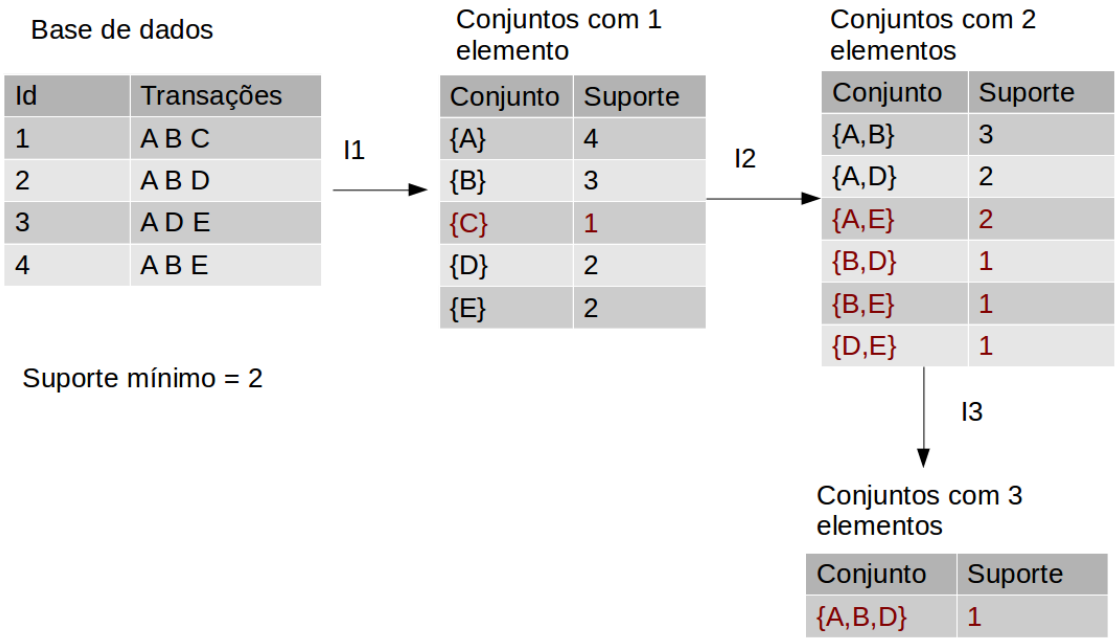
As figuras 3 e 4 contêm uma apresentação do comportamento e um exemplo de execução do algoritmo, respectivamente.

Figura 2: Núcleo do algoritmo Apriori.

1) $L_1 = \{\text{large 1-itemsets}\}$;
2) **for** ($k = 2$; $L_{k-1} \neq \emptyset$; k++) **do begin**
3) $C_k = \text{apriori-gen}(L_{k-1})$; // New candidates
4) **forall transactions** $t \in \mathcal{D}$ **do begin**
5) $C_t = \text{subset}(C_k, t)$; // Candidates contained in t
6) **forall candidates** $c \in C_t$ **do**
7) c.count++;
8) **end**
9) $L_k = \{c \in C_k \mid c.\text{count} \geq \text{minsup}\}$
10) **end**
11) Answer $= \bigcup_k L_k$;

Fonte: Agrawal, Rakesh (2017, p. 489)

Figura 3: Exemplo do algoritmo Apriori.



Fonte: Elaboração do autor.

2.2.1 Bitmap

O índice bitmap, segundo O'Neil (2007), é "tipicamente usado para indexar valores de uma coluna X qualquer, na tabela do banco de dados". Os índices são reorganizados de tal forma que, caso uma linha possua um determinado valor em uma coluna X de uma tabela qualquer, é marcado como verdadeiro para a coluna do índice cujo valor é correspondente ao

valor encontrado, e caso contrário é marcado como falso (CHAN, 1998). Um exemplo de resultado obtido pode ser visto na tabela 1.

Tabela 1: Exemplo de resultado do índice bitmap.

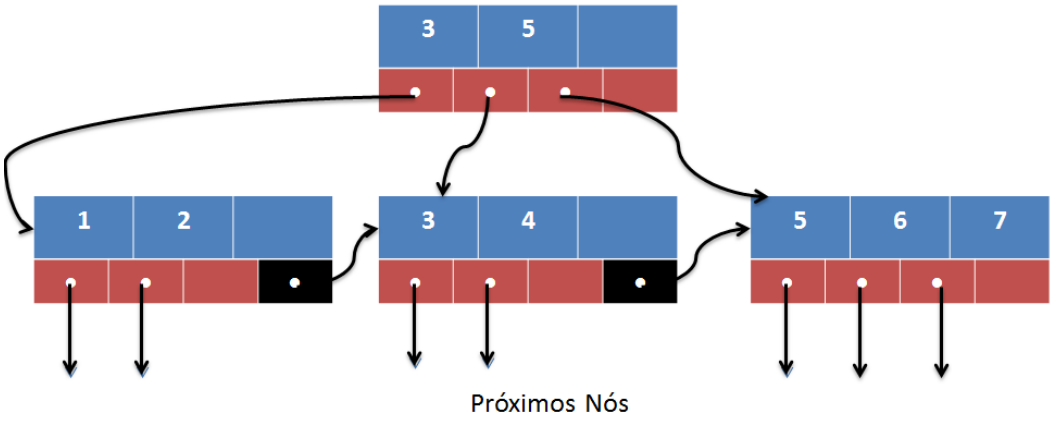
Transação	Índice do valor a	Índice do valor b	Índice do valor c	Índice do valor d
abc	1	1	1	0
abd	1	1	0	1
bcd	0	1	1	1
acd	1	0	1	0

Fonte: Elaboração do autor.

2.2.3 Árvore B+

Segundo Folk (1992), a Árvore B+ é uma estrutura de dados do tipo árvore, cuja vantagem é cada nó possuir um ponteiro para vários nós, e não somente dois nós como na árvore binária. Isto faz com que o tamanho da árvore seja menor que a árvore binária, diminuindo o número de nós que serão percorridos para achar algum valor específico. O construtor da estrutura possui dois parâmetros de entrada: o número máximo e o número mínimo de chaves que cada nó irá guardar.

Figura 4: Exemplo de estrutura da árvore B+.



Fonte: Elaboração do autor.

Cada nó possui o número de valores de entrada e se liga com o nó seguinte correspondente de acordo com a sua entrada. A ligação é feita de forma ordenada em cada nível da árvore, fazendo com que os nós se organizem em uma forma de lista, onde sempre o

último ponteiro aponta para o próximo nó da lista, e o último nó aponta para um nó a mais do próximo nível. O exemplo da estrutura pode ser visto na Figura 5.

Para a criação do bitmap, foi necessário guardar a relação “qual valor equivale a qual coluna na matriz bitmap”. Para isto, foi utilizada uma Árvore B+ para guardar esta relação, pois o tempo de procurar um valor é bem menor que procurar numa lista (FOLK, 1992). No caso da Tabela 1, foi preciso guardar o índice do valor b em cada transação, que equivale a coluna 2 da matriz bitmap.

2.2.4 Apriori baseado em Bitmap

Outra implementação do algoritmo Apriori utiliza o índice bitmap para gerar as combinações e as representações dos conjuntos. Segundo Sandaruwan (2011) e Lin (2003), a representação bitmap gera um ganho de desempenho no momento de realizar a contagem de algum conjunto de item em particular. Em vez de olhar para a tabela, é guardado o número de cada item frequente em cada conjunto de itens. O algoritmo segue os seguintes passos:

No passo 1, é gerado o bitmap da base de dados a qual foi passada como entrada.

No passo 2, são obtidos os candidatos para o próximo passo do algoritmo. Com os conjuntos de itens gerados com tamanho i, é possível gerar conjuntos de itens de tamanho i+1 unindo dois conjuntos de tamanho i, criando a combinação de todos os conjuntos de dados que possuem i+1 a partir dos que possuem tamanho i. Este passo é executado somente na CPU.

No passo 3, é necessário retirar todos os candidatos que possuem o tamanho i+1 que não possui todos os subconjuntos de tamanho i frequentes. Nesta etapa, como é feita uma contagem para verificar os conjuntos mais frequentes, é utilizada a GPU.

No passo 4, é obtido o bitmap dos conjuntos candidatos de tamanho i+1. Para isto, é usada uma operação AND bit a bit, correspondendo ao bitmap de conjuntos com tamanho 1 que geram o conjunto candidato de tamanho i+1. Para alcançar mais paralelismo, é simplesmente realizada uma multiplicação gerando o bitmap resultante dentro da própria GPU, em vez de calcular cada candidato separadamente. O resultado é guardado em uma matriz bidimensional.

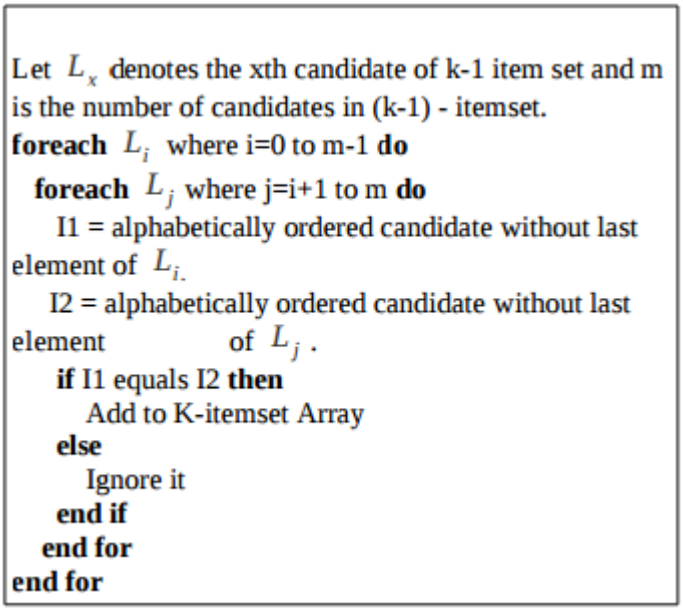
O número de transações com o conjunto pode ser obtido contando o número de 1's no bitmap de transação. Esta parte é totalmente paralelizável, criando um thread para cada item.

Com isto, os itens frequentes de tamanho i+1 são selecionados, comparando com o suporte mínimo, e com isto, os próximos candidatos de tamanho i+2 são gerados a partir dos candidatos i+1 frequentes.

Em seguida o algoritmo volta para o passo 2 e continua o processo para achar os conjuntos de tamanho de um elemento a mais. O algoritmo termina quando não há mais conjuntos disponíveis para processar.

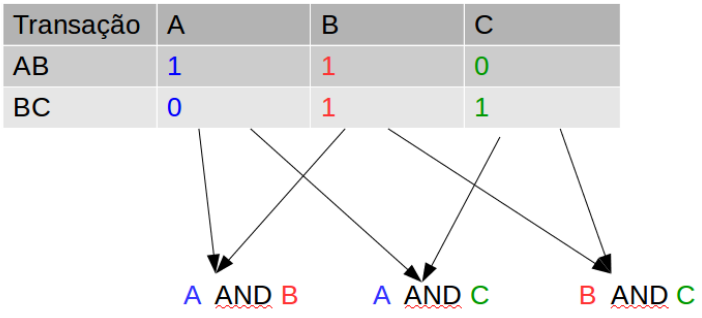
Nas figuras 6 e 7, podemos ver o algoritmo correspondente à geração das chaves candidatas e um exemplo da geração desta matriz, respectivamente.

Figura 5: Geração das chaves candidatas usando Apriori baseado em bitmap.



Fonte: Sandaruwan, Viraj (2011,p. 4)

Figura 6: Exemplo do passo de geração da matriz de conjuntos.



Transação	{A,B}	{A,C}	{B,C}
AB	1	0	0
BC	0	0	1

Fonte: Elaboração do autor.

2.2.4 Considerações utilizadas

Para a execução do apriori bitmap na GPU, é necessário criar o bitmap da base de dados de entrada. A criação deste bitmap foi feita utilizando os seguintes passos: Primeiramente, é criada uma árvore B+ para guardar os valores da base de dados de acordo com a sua coluna correspondente no bitmap. Por exemplo, o valor X qualquer da coluna Y qualquer do dataset equivale a uma coluna Z do bitmap. Para isto, foi utilizado o seguinte algoritmo de acordo com a figura 8:

Figura 7: Algoritmo das associações da base para a árvore.

```
while(getline(fin,str))
{
  str_vector = explode(",",str); // Divindo os campos recebidos pela base

  for(i = 0 ; i < str_vector.size();i++)
  {
    sprintf(num_result_temp_char,"%d",i);
    num_result_temp = num_result_temp_char; // Colocar tag no inicio com a posição da coluna trabalhada
    num_result_temp += "_";
    num_result_temp += str_vector[i];
     // Caso não exista este valor na árvore, acrescentar na árvore
     if(! (b_plus_tree.find(num_result_temp,&resultado_do_find) ) )
    {
        b_plus_tree.insert( num_result_temp ,SSTR(numero_de_items));
        numero_de_items++;
    }
  }
  numero_de_linhas++;
}
```

Fonte: Elaboração do autor.

A tag definida no código apresentado na fig. 8 como sendo “num_result_temp_char”, colocada no início da variável “num_result_temp”, serve para representar que este valor se encontra numa coluna específica, de modo que dado um valor X qualquer numa coluna Y qualquer não tenha a mesma representatividade do valor X em outra coluna diferente de Y, fazendo com que esses valores sejam representados em duas colunas distintas da matriz bitmap.

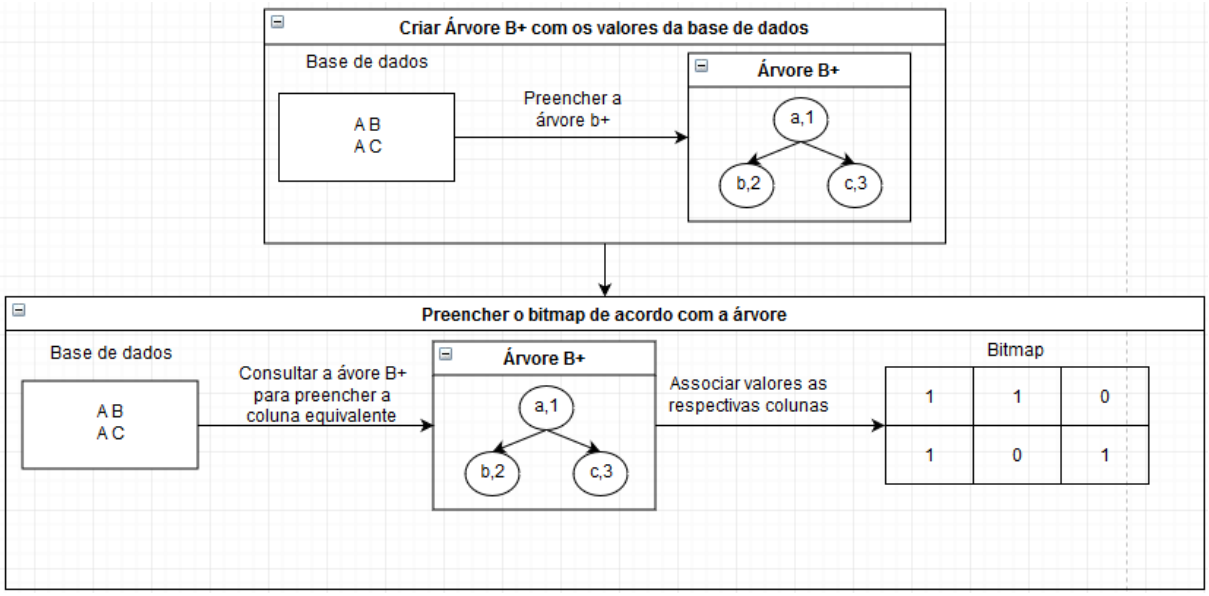
Após a criação da árvore B+, a base de dados foi percorrida novamente, associando os valores na base de dados com a coluna correspondente do bitmap através da árvore criada. Após estes passos a criação do bitmap foi completa. O fluxo completo da criação do bitmap pode ser observado de acordo com a fig 9.

A utilização da árvore B+ para criação do bitmap ao invés de utilizar uma lista para guardar a associação dos valores que aparecem na base de dados com a sua respectiva coluna

no bitmap se deu pelo fato da lista gerar um desempenho muito baixo, principalmente quando era necessário percorrer os últimos valores da lista.

Foi considerado o conceito em grade para realizar o mapeamento do bitmap dentro da placa de vídeo. Com isto, foi possível utilizar a paralelização do Apriori bitmap dentro da GPU. Para a execução do algoritmo, foi utilizado o CUDA para executar em três dimensões, onde a primeira dimensão representou a matriz bitmap do dataset, a segunda dimensão foi a matriz da iteração e a terceira representou a matriz resultante da permutação das colunas entre as duas matrizes.

Figura 8: Fluxo de criação do bitmap



Fonte: Elaboração do autor.

Neste caso, a matriz de iteração considerada era a matriz resultante da iteração anterior. Na primeira iteração, foi considerada a própria matriz do dataset. A realização da operação foi feita utilizando o código que está na figura 10.

Figura 9: Algoritmo do cálculo do bitmap do novo conjunto

```
__global__ void two_list_freq_count_kernel(unsigned int *data1, size_t rows, size_t cols1, size_t size1,
                                           unsigned int *data2, size_t cols2, size_t size2,
                                           int *count, unsigned int *outdata, size_t outsize) {
   size_t ii = threadIdx.x + blockDim.x * blockIdx.x;
   size_t jj = (threadIdx.y + blockDim.y * blockIdx.y) * 32;
   size_t tt0 = (threadIdx.z + blockDim.z * blockIdx.z) * 100;
   int sum[32];
   unsigned int chtmp, chout;
   bool b2;
   if (ii < cols1 && jj < cols2) {
      for(size_t cnt=0;cnt<32;cnt++) {
         sum[cnt]=0;
      }
      for (size_t tt=tt0; tt < tt0 + 100 && tt<rows ; tt++) {
         chout = 0;
         chtmp = __ldg(&data1[(tt*size1 + ii)/32]);
         if ( (chtmp & (unsigned int)(ONEBIT>>(ii%32))) != 0) {
            chtmp = __ldg(&data2[(tt*size2 + jj)/32]);
            chout |= chtmp;
            size_t cnt;
            for (cnt=0; cnt<32; cnt++) {
               b2 = (chtmp & (unsigned int)(ONEBIT>>cnt)) != 0;
               if (b2) {
                  if (jj+cnt < cols2) {
                     sum[cnt]++;
                  } else {

                     chout -=  chout & (ONEBIT>>cnt);
                  }
               }
            }
            atomicOr(&outdata[(tt*outsize + ii*cols2 + jj)/32],
                        chout>>((tt*outsize + ii*cols2 + jj)%32));
         }

      }
      for(size_t cnt=0;cnt<32;cnt++) {
         if (jj+cnt < cols2) {
            atomicAdd(&count[cols2*ii + jj + cnt ], sum[cnt]);
         }
      }
   }
```

Fonte: Elaboração do autor.

3 METODOLOGIA

3.1 AMBIENTE

O computador utilizado possui um processador Intel core i7 3770k com o clock de 3.2 GHz, 8 GB de memória RAM DDR3 e uma placa de vídeo Geforce (NVIDIA, 2017) da fabricante ZOTAC, cujo modelo é “980 ti AMP!”, a qual possui 6 GB de memória GDDR5.

Foi utilizada a linguagem C++ com a biblioteca CUDA Toolkit 9 (NVIDIA, 2017) no desenvolvimento do algoritmo.

O sistema operacional utilizado foi Linux com a distribuição Ubuntu 11.04.

3.2 BASE DE TESTES

Foram utilizadas quatro bases de testes para fazer a análise de desempenho: a “íris”, a “connect”, a “renda do censo”, e a “mushroom”, todas obtidas no repositório da UCI (DUA; KARRA TANISKIDOU, 2017). As bases de testes foram analisadas e, a partir disso, foram obtidas as seguintes informações, de acordo com a tabela 2:

Tabela 2: Informações obtidas das bases

Banco	Número de Transações	Tamanho das transações	Número de itens diferentes
Iris Dataset	150	5	100
Connect	67557	43	129
Censo	48842	14	22146
Mushroom	8124	23	119

Fonte: Elaboração do autor.

3.2.1 Iris

A base de dados da íris possui informações de várias plantas do gênero íris. Essa base possui o tamanho de 4,6 KB no formato de arquivo txt, cujo separador é o espaço entre as colunas. Possui 150 registros com cinco atributos cada: comprimento da sépala, largura da sépala, comprimento da pétala, largura da pétala, e a classe da planta. É uma boa base para classificar as classes da planta, pois a partir das quatro métricas que a base possui é possível identificar a classe da planta.

3.2.2 Connect-4

Esta base de dados possui informações a respeito do jogo “connect-4”. Ela possui todos os estados possíveis para o jogo, num tabuleiro 8x8, onde nenhum dos jogadores ainda ganhou e o próximo movimento não é forçado para a finalização do jogo (por exemplo: no próximo turno o jogador irá colocar a peça onde irá fazer uma linha com 4 peças). A base possui o tamanho de 6 MB no formato .data, com valores separados por vírgula. Conforme especificado pelo repositório da UCI (DUA; KARRA TANISKIDOU, 2017), os 42 primeiros atributos representam as posições preenchidas no tabuleiro, verticalmente de baixo para cima com os valores “x”, “o” e “b”, onde “x” representa o primeiro jogador, “o” representa o segundo jogador e “b” indica que a posição não foi obtida por nenhum dos jogadores. O último atributo representa se o primeiro jogador ganhou, perdeu ou empatou o jogo.

3.2.3 Renda do Censo

Segundo o repositório da UCI (DUA; KARRA TANISKIDOU., 2017), esta base de dados foi feita em cima de uma extração do banco de dados do censo de 1994. A ideia principal desta base de dados é descobrir as características das pessoas que influenciam elas possuírem uma renda acima de 50 mil dólares em um ano. Possui 4MB de tamanho, em formato .data, cujos dados são separados por vírgula seguido de espaço. A base também possui 48.842 registros, com 14 atributos cada. Os atributos são definidos como: ano, classe trabalhadora, peso na amostra, nível escolar, número do nível escolar (quanto maior o nível escolar maior o nível escolar da pessoa), estado civil, ocupação, relacionamento, raça, sexo, ganho capital, capital gasto, horas por semana, país nativo e o ganho anual (definido somente se é acima de 50 mil ou abaixo de 50 mil dólares).

3.2.3 Mushroom

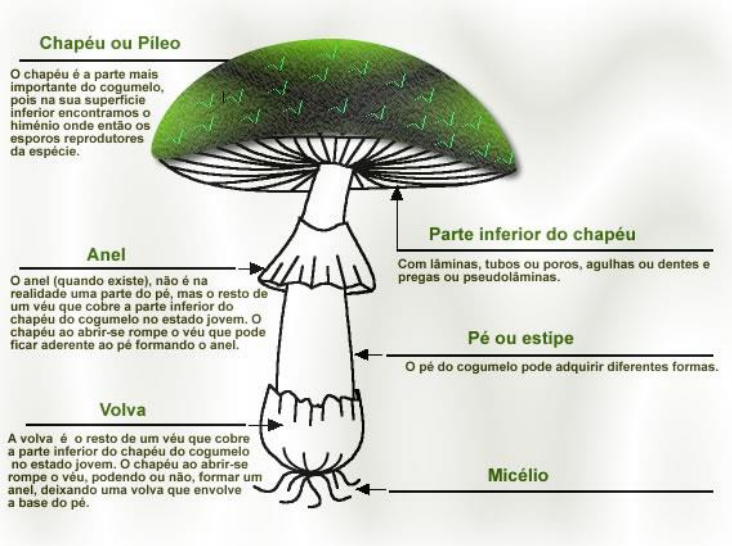
Esta base disponibilizada pela UCI (DUA; KARRA TANISKIDOU, 2017) possui a descrição de amostras hipotéticas a respeito de 23 espécies de cogumelos lamelados que pertencem a duas famílias: Agaricus e Lepiota. O objetivo desta base é tentar identificar que dadas às características do cogumelo, se ele é venenoso ou comestível. Esta base possui 373,7

KB de dados, no formato .data, utilizando vírgula para separar as colunas. A base também possui 8124 registros com 23 atributos, onde o primeiro é utilizado como critério de classificação, com os valores “p” para venenoso e “e” para comestível.

Os outros 22 atributos definem as características dos cogumelos, sendo elas: a forma do chapéu, a forma da superfície, a cor do chapéu, se a amostra possui contusões, o cheiro, o anexo da lamela (ou lâmina na parte inferior do chapéu), o espaçamento das lamelas, o tamanho das lamelas, a forma do pé, a raiz do pé, a superfície do pé acima do anel, a superfície do pé abaixo do anel, a cor do pé acima do anel, a cor do pé abaixo do anel, o tipo do véu, a cor do véu, o número de anéis, o tipo de anel, a cor da esporada, o tipo de população e o habitat do cogumelo.

A esporada é um diagnóstico que revela a cor dos esporos quando são vistos em grande quantidade (EVENSON, 1997). Para a melhor compreensão, as partes do cogumelo podem ser vistas na figura 11.

Figura 10: Partes de um cogumelo.



Fonte: Martinez, Marina (2017)

3.3 IMPLEMENTAÇÃO

Em ambos os programas foi criado o bitmap da base de dados no início da execução do algoritmo, utilizando somente a CPU, gerando todos os valores possíveis da base de dados, adicionando em uma lista. A lista representa as colunas da matriz bitmap para o valor específico, e as linhas representam a transação.

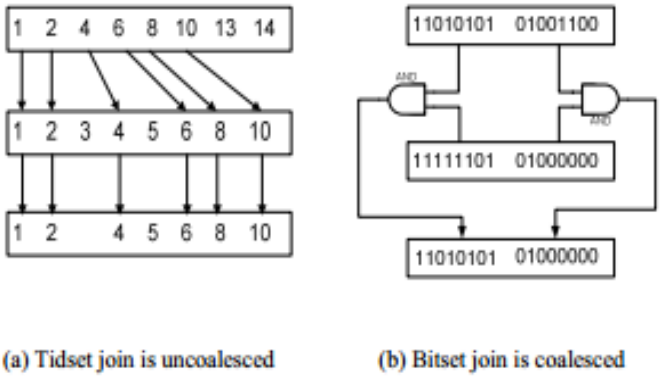
A implementação da versão CPU do Apriori bitmap foi a primeira a ser criada. O algoritmo lê o índice bitmap da base, criado no início da execução do algoritmo, e responde com os conjuntos mais frequentes a partir de um suporte mínimo definido na entrada do programa.

Após isto, foi feita uma implementação do mesmo algoritmo, utilizando a linguagem CUDA. Para facilitar o desenvolvimento do algoritmo foi utilizada a biblioteca CUB (MERRILL, 2013), a qual ajuda a manipulação dos threads na execução nas etapas de criação de uma nova matriz a partir dos candidatos menores e contagem dos candidatos.

Após a implementação de ambas as versões, foi feita uma tentativa de obter um ganho de desempenho na geração do bitmap. Em vez de utilizar uma lista para representar os valores, foi feita uma árvore B+, a qual geraria um desempenho maior na busca de qual coluna da matriz receberia o valor positivo.

Com o intuito de melhorar ainda mais o desempenho do bitmap, foi feita uma mudança na estrutura do bitmap, a fim de gerar uma matriz condensada, em vez de uma matriz normal. Com isto a matriz teria um tamanho bem menor, consumindo menos memória e facilitando o join. Esta matriz foi gerada seguindo o mesmo princípio da implementação do GPapriori feito por (ZHANG, 2011). Esta implementação pode ser vista na figura 12.

Figura 11: Comparação entre o bitmap normal e o não condensado.



Fonte: Zhang, Fan (2011, p. 591)

4 RESULTADOS

Para análise de desempenho de ambos os algoritmos, foram analisados os tempos totais de execução em cada uma das bases. Em seguida, foi analisado o tempo de criação do primeiro bitmap utilizando a lista e a árvore B+ para critério de ganho de desempenho. Para o algoritmo que utiliza a GPU, também foram analisados os tempos de alocação de memória na GPU. Dado o tempo de execução ser muito grande, o algoritmo foi executado até achar os conjuntos de, no máximo, dois elementos nas bases de dados “mushroom” e “connect”. Para a análise dos tempos gastos com processos na GPU, foi utilizado o programa “nvprof”, disponibilizado junto com o framework CUDA da Nvidia.

4.1 TEMPO DE EXECUÇÃO TOTAL

Primeiramente, foram executadas ambas as implementações com suporte mínimo igual a 10 e até encontrar conjuntos de tamanhos 10 (exceto para o dataset da íris). Foram obtidos os seguintes tempos, de acordo com a Tabela 3.

Tabela 3: Tempo total das execuções dos algoritmos utilizando matriz esparsa, árvore B+ e com suporte 10.

Base	Tempo de CPU	Tempo de CPU+GPU
Iris dataset	00:00:02,14	00:00:00,35
Mushroom	109:34:56	00:00:02,26
Connect	134:23:23	00:00:05,82
Censo	33:41:43	00:00:23,4

Fonte: Elaboração do autor.

A partir dos resultados encontrados, o ganho de desempenho chegou a mais de 1000% em vários casos. Estes resultados demonstram que a utilização da GPU na execução do algoritmo é realmente uma vantagem para o desempenho.

Também no resultado obtido, foi possível verificar que o tamanho do dataset influenciou de forma significativa o custo de execução do algoritmo para a GPU, possivelmente pela transferência de dados. Foi identificado que em datasets menores o tempo de execução na CPU era maior em função do tamanho dos conjuntos encontrados, resultando em um maior número de iterações do Apriori no dataset. Este custo de iteração foi bem menor

na GPU, fazendo com que o algoritmo executasse de maneira mais rápida nestes casos, onde o tamanho do dataset era menor e o número de iterações do Apriori era maior.

4.2 TEMPO DE CRIAÇÃO DA PRIMEIRA MATRIZ BITMAP

O tempo de criação do bitmap foi bastante reduzido com a utilização de uma árvore B+. O tempo de criação do primeiro bitmap chegou a ser reduzido em torno 50% (Tabela 4) em todos os casos, com o uso desta estrutura em vez de utilizar uma lista para armazenar os itens diferentes.

Tabela 4: Tempo gasto para criação da matriz bitmap inicial

Base	Tempo utilizando lista (seg)	Tempo utilizando árvore B+ (seg)
Iris dataset	0,0031	0,0014
Mushroon	0,4	0,2121
Connect	6,1	3,3233
Censo	87,2148	28,9804

Fonte: Elaboração do autor.

4.3 TAMANHO DOS BITMAPS GERADOS

Nesta etapa, medimos o tamanho de cada bitmap que foi gerado para averiguar os tempos consumidos posteriormente. Além do tamanho em número de linhas e colunas, foi necessário identificar o tamanho ocupado pela matriz bitmap em bytes (Tabela 5).

Tabela 5: Tamanho da matriz bitmap inicial

Base	Tamanho em bytes	Número de linhas	Número de Colunas
Iris dataset	2368	150	100
Mushroon	119832	8124	119
Connect	644658813	67557	129
Censo	721095906	48842	22146

Fonte: Elaboração do autor.

4.4TEMPO DE ALOCAÇÃO DO PRIMEIRO BITMAP NA GPU

Nesta etapa, foi medido somente o tempo da criação do primeiro bitmap na GPU, conforme apresentado na Tabela 6, para verificar a utilização das matrizes condensadas.

Tabela 6: Tempo de alocação do primeiro bitmap na GPU

Base	Tempo não utilizando matriz esparsa (em ms)	Tempo esparsa utilizando matriz esparsa (em ms)
Iris dataset	134,64	141,19
Mushroon	157,63	133,46
Censo	1287,076	299,32
Connect	803,14	114,75

Fonte: Elaboração do autor.

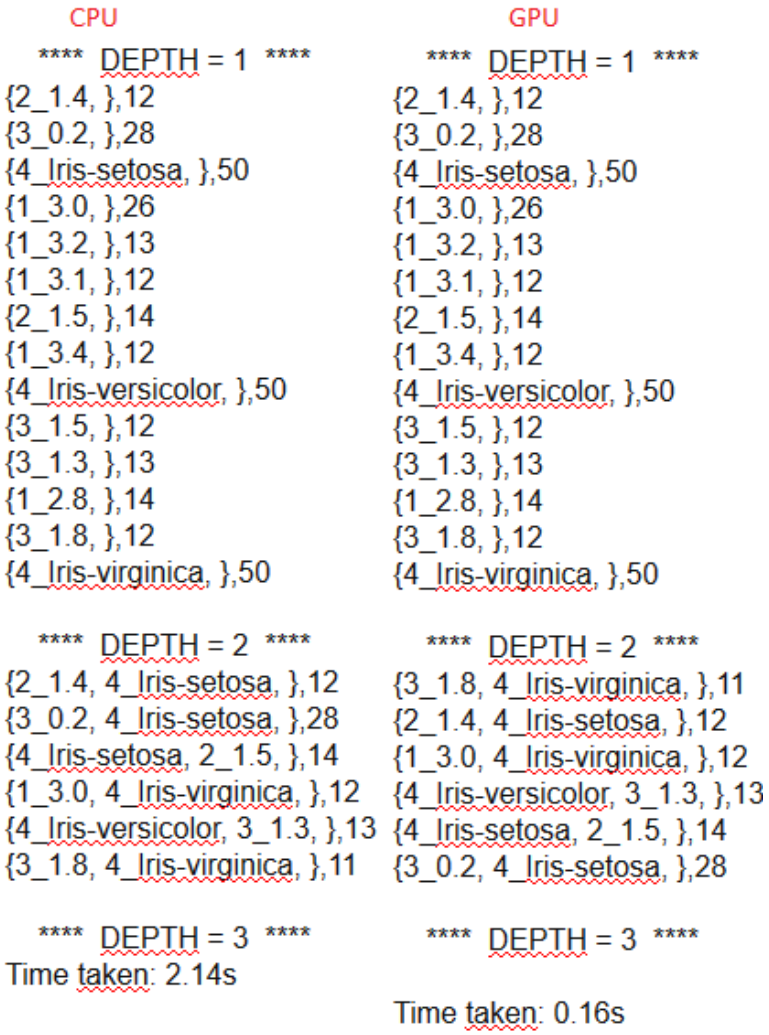
A utilização de matrizes esparsas ajuda bastante no tempo de alocação da memória na GPU, tendo um ganho de quase 700% na velocidade de alocação.

4.5 COMPARAÇÃO DA SAÍDA EM AMBAS AS APLICAÇÕES

Em ambas as aplicações, as saídas obtidas foram iguais em todos os casos. Apesar de o algoritmo ter sido modificado para encaixar na arquitetura das GPUs, o resultado final foi o mesmo, mantendo assim, o objetivo do algoritmo Apriori original.

Uma comparação dos resultados pode ser vista na figura 13, utilizando pelo próprio resultado do dataset íris com suporte mínimo 10.

Figura 12: Comparação da saída da CPU vs GPU.



Fonte: Elaboração do autor.

5 CONCLUSÃO

A utilização de GPUs se mostrou bastante adequada para a solução do problema, possibilitando um alto grau de paralelização e acelerando a execução do algoritmo. Entretanto, para a implementação do mesmo, o algoritmo teve que ser modificado, para atender ao paradigma de programação paralela.

A partir dos resultados obtidos, o desempenho do algoritmo Apriori utilizando GPU foi muito melhor que utilizando uma CPU convencional. Outra mudança foi a utilização de estrutura de dados, possibilitando o aumento do desempenho do algoritmo executado na CPU.

Trabalhar com GPUs exige o conhecimento da arquitetura para maximizar o ganho de desempenho, dadas as particularidades do mesmo, fazendo com que seja necessário modificar os algoritmos utilizados para obter o grau de paralelismo ao qual a GPU oferece.

Este trabalho foi uma boa oportunidade para aprender a programar utilizando uma arquitetura diferente do processador comum, principalmente um ambiente com GPUs da NVIDIA com suporte à linguagem CUDA.

5.1 TRABALHOS FUTUROS

É possível identificar a utilização de mais de uma GPU em paralelo e comparar o ganho de desempenho do algoritmo com os resultados atuais. Duas abordagens podem ser validadas: uma utilizando o mesmo algoritmo, e outra testando armazenamento de dados distribuído.

Outro trabalho seria verificar como resolver o problema do Apriori bitmap quando for necessário consumir mais memória na GPU do que o permitido. A biblioteca atual CUDA não permite alocar uma memória maior que o tamanho disponível dentro da própria GPU, e não foi encontrado nenhum tipo de funcionalidade que permita efetuar swap da memória. Uma ideia seria utilizar uma estrutura de dados que proporciona a execução parcial da base de dados e efetuar a contagem dos itens apenas para um pedaço da base de dados.

REFERÊNCIAS

AGRAWAL, Rakesh; SRIKANT, Ramakrishnan. **Fast algorithms for mining association rules**. 1994. Disponível em: <http://www.vldb.org/conf/1994/P487.PDF>. Acesso em: 5 abr. 2017.

CHAN, Chee-Yong; IOANNIDIS, Yannis E. **Bitmap index design and evaluation**. 1998. Disponível em: <http://www.comp.nus.edu.sg/~chancy/sigmod98.pdf>. Acesso em: 12 abr. 2017.

DUA, D.; KARRA TANISKIDOU, E. **UCI Machine Learning Repository**. 2017. Disponível em: http://archive.ics.uci.edu/ml. Acesso: 07 abr. 2018

EVENSON, Vera Stucky. **Mushrooms of Colorado**: and the southern Rocky Mountains. Denver: Denver Botanic Gardens, 1997.

FANG, Wenbin et al. **Parallel data mining on graphics processors**. 2008. Disponível em: <https://pdfs.semanticscholar.org/4e32/62ef632f251b2e1a22c0ab9ca8a3356c4203.pdf>. Acesso em: 24 mar. 2017.

FANG, Wenbin et al. **Frequent itemset mining on graphic processors**. 2009. Disponível em: <http://www.comp.nus.edu.sg/~hebs/pub/wenfrequent_sigmod09.pdf>. Acesso em: 27 mar. 2017.

FOLK, Michael J.; ZOELLICK, Bill; RICCARDI, Greg. **File structures**: an object-oriented approach with C++. Reading: Addison-Wesley, 2006.

LIN, Tsau Young; HU, Xiaohua; LOUIE, Eric. A fast association rule algorithm based on bitmap and granular computing. In: IEEE INTERNATIONAL CONFERENCE ON FUZZY SYSTEMS, 12., 2003, St. Louis. **Proceedings**... New York: IEEE, 2003.

MARTINEZ, Marina. **Cogumelo**. 2017. Disponível em: <https://www.infoescola.com/reino-fungi/cogumelo/>. Acesso em: 5 abr. 2017. (mudei a entrada)

MERRILL, D. **CUB**. 2013. Disponível em: <http://nvlabs.github.io/cub/>. Acesso em: 5 abr. 2017. (mudou o ano)

NVIDIA. **Cuda C programming guide**. Disponível em: http://docs.nvidia.com/cuda/cuda-c-programming-guide/, 2017. Acesso em: 05 Abr. 2017

NVIDIA. **Geforce 980 ti specifications**. Disponível em: http://www.geforce.com/hardware/desktop-gpus/geforce-gtx-980-ti/specifications. Acesso em: 05 Abr. 2017

O'NEIL, Elizabeth; O'NEIL, Patrick; WU, Kesheng. Bitmap index design choices and their performance implications. In: INTERNATIONAL DATABASE ENGINEERING AND APPLICATIONS SYMPOSIUM, 11., 2007, Banff. **Proceedings**... New York: IEEE, 2007.

SANDARUWAN, Viraj et al. **A bitmap based apriori algorithm on graphic processors**. 2011. Disponível em: <https://www.researchgate.net/publication/261003644_A_bitmap_based_apriori_algorithm_on_Graphic_processors>. Acesso em: 5 abr. 2017.

SILVA, Marcelino Pereira dos Santos. **Mineração de dados**: conceitos, aplicações e experimentos com Weka. 2004. Disponível em: <http://www.lbd.dcc.ufmg.br/colecoes/erirjes/2004/004.pdf >. Acesso em: 2 abr. 2017.

SUJEETH, Arvind K. et al. OptiML: an implicitly parallel domain-specific language for machine learning. In: INTERNATIONAL CONFERENCE ON MACHINE LEARNING, 28., 2011, Bellevue. **Proceedings**... New York: ACM, 2011. p. 609-616.

ULLMAN, J. D. **A survey of Association-Rule Ming**. 2000. Disponível em: <https://link.springer.com/content/pdf/10.1007/3-540-44418-1_1.pdf>. Acesso em: 8 abr. 2017.

ZHANG, Fan; ZHANG, Yan; BAKOS, Jason. Gpapriori: Gpu-accelerated frequent itemset mining. In: IEEE INTERNATIONAL CONFERENCE ON CLUSTER COMPUTING, 2011, Austin. **Proceedings**... New York: IEEE, 2011.