OTIMIZAÇÃO DO ALGORITMO EXPECTATION MAXIMIZATION PARA O MODELO DE MISTURAS GAUSSIANAS USANDO CUDA

Elaynne Torres, Hendrik Macedo, Marco Chella, Leonardo Matos
Departamento de Computação/Universidade Federal de Sergipe, Brasil
elaynne.ufs@gmail.com, hendrik@dcomp.ufs.br, marco@dcomp.ufs.br, leonardo@dcomp.ufs.br

***Resumo*: Este artigo descreve técnicas de otimização realizadas no algoritmo *Expectation Mazimization* (EM) para o treinamento de Modelo de Misturas de Gaussianas (GMM) utilizando a arquitetura CUDA disponível em GPUs da NVIDIA. O objetivo é o de reduzir o tempo de processamento deste treinamento em aplicações que façam uso do reconhecimento de fala contínuo (*Continuous Speech Recognition* - CSR) para a língua portuguesa do Brasil. Os experimentos foram conduzidos em 5 (cinco) diferentes bases de dados e variados tamanhos de gaussianas. Resultados mostram redução de quase 90% do tempo de execução E-passo de treinamento se comparado com trabalhos anteriores.**

***Palavras-chave:* CUDA; GPU; Expectation Maximization; Modelo de Misturas de Gaussianas; Reconhecimento automático de fala contínuo.**

***Abstract*: This paper describes optimization techniques performed on the Expectation Maximization (EM) algorithm on NVIDIA's CUDA architecture. The goal is to improve time performance of Gaussian Mixture Model (GMM) training for Continuous Speech Recognition (CSR) for the Portuguese language. Experiments have been made with 5 (five) different datasets and varying number of Gaussians. Results show a reduction of almost 90% on the E-step runtime if compared to previous work.**

***Keywords:* CUDA; GPU; Expectation Maximization; Gaussian Mixture Models; Continuous Speech Recognition.**

I. INTRODUÇÃO

Acelerar as etapas de algoritmos presentes no aprendizado de máquina permite criar sistemas de reconhecimento automático de fala que executem em tempo real. O objetivo deste artigo é contribuir na construção de um Modelo Acústico para Reconhecimento Automático de Fala da Língua Portuguesa do Brasil. Um sistema CSR (Continuous Speech Recognition) consiste em identificar com precisão todos os fonemas pronunciados por usuários. Portanto, deve-se mapear o sinal acústico originário para uma sequência de palavras que seguem um conjunto de regras sintáticas baseadas na língua em questão.

O reconhecimento automático de fala é realizado em duas fases. Na fase de treinamento, usam-se algoritmos de aprendizado de máquina, onde o sistema memoriza um conjunto de modelos de referência. Na fase de teste/validação, o sistema compara um sinal de fala com todos os modelos de referência e retorna aquele mais próximo como o padrão reconhecido. Nota-se que a precisão do desempenho melhora quando há mais modelos de referência para se comparar com o padrão. No entanto, o tempo para encontrar o padrão mais próximo aumenta exponencialmente com um conjunto maior de modelos de referência. Este tempo pode ser reduzido por paralelização de rotinas computacionalmente dispendiosas em uma unidade de processador gráfico (GPU) [1].

A arquitetura CUDA™ é um novo framework disponível para GPUs NVIDIA que permitem o desenvolvimento de códigos que executem de forma paralela [2]. Ela funciona como uma única instrução de múltiplos dados do sistema (SIMD), onde cada multiprocessador é capaz de processar apenas uma instrução por vez. O elemento básico de processamento paralelo é um thread, como a CPU, o bloco e o grid. Tanto o grid e os blocos podem ser uni-, bi- ou tridimensionais. Uma chamada de kernel precisa especificar as dimensões do grid e blocos e, assim, é possível executar kernels com diferentes arranjos de threads dentro do mesmo aplicativo [3].

A hierarquia de memória consiste em registradores, memória local, memória compartilhada, memória global e memória constante. Os registradores são a memória mais rápida. Memória local é privada para cada segmento. A memória compartilhada é mais lenta e menor do que a memória local, mas é acessível por todos os threads do mesmo bloco, permitindo que os threads trabalhem de forma colaborativa em uma única corrida. A memória global é a maior e mais lenta das memórias da GPU, mas é acessível por qualquer segmento, independentemente do bloco, permitindo assim que diferentes kernels compartilhem dados comuns. A memória constante é um processo lento, read-only memory, usada para armazenar dados que não modificam [4].

Existem algumas vantagens no uso de CUDA: o código GPU pode acessar endereços diferentes na memória e a disponibilidade de memória compartilhada. Isto ajuda a reduzir acessos da memória global e, como resultado, proporciona uma maior largura de banda. No entanto, existem algumas limitações. Uma delas é que a transferência de dados entre a GPU e a memória do host (CPU) é mais lenta que as transferências dentro da memória da GPU [3].

Neste artigo é apresentada a implementação otimizada paralelizada na linguagem CUDA C do algoritmo Expectation Maximisation (EM) de estimação dos modelos de misturas gaussianas (GMM). O GMM é um dos modelos estatísticos mais utilizados para tarefas de aprendizado de máquina, sendo o modelo paramétrico mais flexível. A otimização do algoritmo EM para GMM permitirá o desenvolvimento de aplicações futuras e aceleradas com reconhecimento automático de fala.

Trabalhos anteriores apresentam diferentes abordagens para prover paralelização do algoritmo EM. Em [5], os autores

apresentam uma estratégia para acelerar o algoritmo EM utilizando redução de domínio. O EM é usado para reconstruir volumes 3D a partir de imagens ruidosas de Electron Cryomicroscopy de partículas macromoleculares. Em [6] os autores implementam uma versão CUDA do algoritmo EM para verificação de locutor baseado no Gaussian Mixture Modeling-Universal Background Modeling (GMMUBM). Eles utilizaram dois kernels para o EM, um para o E-step e um para o M-Step. Esses trabalhos focam em problemas não relacionados á estimação de Gaussian Mixture Models (GMM), o qual estamos particularmente interessados. Em [7], os autores derivam um método algoritmico para aprendizado de GMM incremental a partir de um teste de hipótese e algoritmo baseado em fusão. A parte de maior consumo do algoritmo é acelerada por GPU. O trabalho, entretanto, não utiliza EM.

Neste artigo, nós propomos especificamente a otimização de uma abordagem multi-kernel para a paralelização do algoritmo EM para a estimação do GMM. Nesse sentido, nosso trabalho se assemelha à [8] e [9], sendo uma otimização do trabalho descrito em [3].

O algoritmo EM para treinamento de GMM é detalhado na seção 2. A seção 3 traz a metodologia adotada para experimentação: paralelização das etapas do E-passo, descrição da otimização aplicada, descrição das bases de dados utilizadas. Os resultados destes experimentos estão apresentados e discutidos na seção 4. A seção 5 traz a conclusão do trabalho.

II. Revisão Teórica

A. *Expectation Maximization (EM)*

Os modelos estatísticos são utilizados em muitas técnicas de aprendizado de máquina. O expectation-maximization (EM) permite a aprendizagem de parâmetros que regem a distribuição dos dados da amostra com algumas características em falta. O algoritmo EM para GMM é um algoritmo baseado em cálculos estatísticos que computam a estimativa de parâmetros de máxima verossimilhança nos casos em que as equações não podem ser resolvidas diretamente onde o modelo depende de variáveis latentes não observados. O princípio geral do algoritmo EM é baseado na maximização da função de probabilidade dada pela equação (1).

$$p(D|\theta) = \prod_{i=1}^{n} p(xi|\theta) \quad (1)$$

onde, D = {*x1*, ..., *xn*} representa o conjunto de dados, θ, os parâmetros da função de densidade de probabilidade que modela o seu comportamento e *xi*, uma observação. A função de verossimilhança, portanto, mede o nível de alinhamento entre o modelo e os dados.

O algoritmo EM é constituído por duas etapas. O E-passo calcula a expectativa condicional da probabilidade logarítmica, condicionalmente ao conjunto de dados observados x e o valor atual dos parâmetros θ:

$$Q(\theta ; \theta^{i}) = E[\mathrm{Ln}\ p(D; \theta) \mid D, \theta^{i}] \quad (2)$$

O M-passo calcula o (i +1)-ésimo parâmetro do vetor θ que maximiza $Q(\theta^{i+1}, \theta)$, dado por:

$$\theta^{i+1} = \arg\max\ Q(\theta; \theta^{i}) \quad (3)$$

O algoritmo é iniciado a partir de um θ (geralmente definido arbitrariamente, embora existam abordagens para optimizar esta escolha) e interage através de ambas as etapas, até que um critério de parada seja satisfeito. O critério utilizado é a variação entre as etapas Q, definidos como:

$$\|Q^{t+1} - Q^{t}\| \leq \mathcal{E} \quad (4)$$

deixando t ser um contador de interação, e ε um critério de convergência pré-definido.

O classificador gaussiano é um dos Modelos Ocultos de Markov (HMM) aplicados ao reconhecimento de fala. No entanto, este método tem limitações ao lidar com dados não-gaussianos, já que suas funções discriminantes são lineares ou quadráticas. Uma solução alternativa é combinar funções de distribuição de probabilidade (pdf). Com efeito, esta aproximação é amplamente usada porque é um método paramétrico que pode ser aplicado para os problemas de classificação não-lineares. Esta técnica é conhecida como modelo de misturas finitas e a sua função de distribuição de probabilidade para uma variável aleatória Y é definida como:

$$p(\check{y};\theta) = \sum_{k=1}^{g} \pi_k\, p(\check{y};\theta) \quad (5)$$

satisfazendo

$$\pi \geq 0,\ k = 1,...,g \text{ and } \sum_{k=1}^{g} \pi_{k=1} \quad (6)$$

onde, g é o número de componentes (pdf) da mistura; πk é a probabilidade dos componentes (vulgarmente conhecidos como mistura de probabilidades), de tal modo que p(˜y; θj) é o pdf do componente em relação aos parâmetros θj.

B. *Modelo de Mistura Gaussiana (GMM)*

Quando usamos modelos de gaussianas, cada componente assume uma distribuição normal multivariada, onde θj = {μj; Σj}. Este modelo é conhecido como Modelo de Mistura Gaussiana (GMM). A equação (5) pode, assim, ser reescrita como:

$$P(\check{y},\mu,\ \Sigma) = \sum_{k=1}^{g} \pi k\ N\left(\check{y}, \mu, \Sigma_k\right) \quad (7)$$

onde Σ é a matriz de covariância e μ representa a média de gaussianas.

Tipicamente, os parâmetros dos componentes do GMM são estimados utilizando o algoritmo EM descrito anteriormente. Para o GMM, os passos de EM são definidos como se segue.

- E-passo. Calcula para cada i dado:

(8) $$W_{ij} = \frac{\pi j N(xi;\pi j,\sum_i)}{\sum_{k=1}^{n} \pi k N(xi;\pi k,\sum_k)}$$

Onde, πj, μj e Σj são os pesos, médias de gaussianas e matrizes de covariância de componente j na etapa t.

- M-passo. Para cada dado j, atualizar os parâmetros:

(9) $$\pi_j = \frac{1}{n}\sum_{i=1}^{n} w_{ij}$$

(10) $$\mu_j = \frac{\sum_{i=1}^{n} w_{ij} x_j}{\sum_{i=1}^{n} w_{ij}}$$

(11) $$\Sigma_j = \frac{\sum_{i=1}^{n} w_{ij}(x_i - \mu_j)(x_i - \mu_j)^T}{n\sum_{i=1}^{n} w_{ij}}$$

O algoritmo EM itera até a convergência do modelo de probabilidade (critério de parada). Para um grande conjunto de dados, o tempo do processo de treinamento pode ser enorme, especialmente em casos em que há um elevado número de componentes. Apesar dessa limitação, os cálculos realizados para cada dado são independentes e, portanto, totalmente paralelizáveis, permitindo a redução desse tempo de treinamento.

Para aferir a performance das implementações paralelizadas em relação às suas versões sequenciais, iremos utilizar a métrica *Speedup*, definida como:

(12) $$Sp = T1 / Tp$$

onde p é o número de processadores da placa gráfica em questão, T1 é o tempo de execução do algoritmo sequencial e Tp é o tempo de execução da versão paralelizada [2], [4].

C. *Algoritmo EM para GMM*

O algoritmo EM para GMM é dividido em duas etapas: a etapa E, de estimação da matriz de probabilidade e a etapa M, de maximização dos pesos, média, e matriz de covariância. O algoritmo recebe como entrada as amostras do conjunto de dados, o número de gaussianas a serem estimados e do limiar como o critério de parada. A cada iteração, o algoritmo inicializa os parâmetros de cada gaussiana (π peso, Σ covariância e μ média). Em seguida, para cada amostra, estima as probabilidades para cada gaussiana e as normaliza. Finalmente, os parâmetros da gaussiana são reavaliados usando os valores de probabilidade. Iterações ocorrem até que o critério de parada seja satisfeito (Algoritmo 1).

Existem três kernels implementados que são responsáveis pela etapa de estimação da matriz de probabilidade:

- Kernel *Mul*. Multiplica o vetor X (samples) pelo inverso da matriz de covariância S^-1 e por X', para o cálculo da função g(x) usada nos p-kernel e pn-kernel.
- *p-kernel*. Para cada um dos componentes da gaussiana j, calcula a probabilidade de cada xi dados condicional para parâmetros θj, multiplicada pelo peso do componente πj. Neste kernel, os blocos de threads são dispostos num grid a j x m, onde m blocos de linha j são responsáveis pelo cálculo da componente j.
- *pn-kernel*: Para cada xi dados, normaliza suas probabilidades calculadas no kernel anterior para cada componente j. Trata-se dos valores de wij. Neste passo, m blocos de segmentos são utilizados e cada bloco é responsável por normalizar as probabilidades para um determinado dado de cada vez, até que toda a base de probabilidade é normalizada.

Cada thread, uma por bloco, estima a probabilidade de uma amostra j na gaussiana i, de acordo com a posição (i, j) do seu bloco no grid (Algoritmo 2). O conjunto de threads no bloco executa a normalização das probabilidades dos valores de uma amostra, utilizando a técnica de redução (Algoritmo 3). Um kernel de 3 dimensões calcula diretamente da memória global a diferença entre o vetor de amostras e as médias e, em seguida, multiplica pela matriz de covariância inversa, alocando numa única memória compartilhada 2D (i,j). Em seguida, é realizada a soma para redução e o resultado é colocado na matriz de probabilidade (Algoritmo 4).

Algoritmo 1: EM para estimação do GMM

Input: samples, $samples_{num}$, $Gaussian_{num}$: $Threshold_{min}$
Output: πI, μi, $\sum_j | i \in \{1,2,\ldots, Gaussian_{num}\}$

```
for i←1, Gaussian_num do
    Initialize parameters (π_i, μ_i, ∑_i);
while-stop condition () do
    for j ←1, Samples_num do
        for i←1, Gaussian_num do
            likelihood_ij← Calculatelikelihood(Sample_j, π_i, μ_i, ∑_i);
        likelihoodj←Normalize Likelihood (likelihoodsj);
    for i←1, Gaussian_num do
        π_i ←Update Weight (likelihood_j);
        μ_i ←Update Mean (likelihoods_j, Samples, π_i);
        ∑_i ←Update Covariance(likelihoods_j, Samples, π_i, μ_i);
```

Algoritmo 2: CUDA p-kernel paralelo

Input: samples, samplesnum, π_i, μ_i, $\sum_i$
Output: likelihoods

```
i←Block Index. Y;
j←Block Index. X;
likelihood_ij←π_i × N(samples_j, μ_i, ∑_i);
```

Algoritmo 3: CUDA pn-kernel paralelo

Input: liklelihoods, Samplesnum, $Gaussian_{num}$
Output: likelihoods

```
i←ThreadIndex.X;
```

Identify applicable sponsor/s here. If no sponsors, delete this text box

```
j←BlockIndex.X;
cachei ←likelihood_ij;
SynchronizeThreads();
Limit ←ThreadsPerBlock/2;
While limit≠0 do
        If i < limit then
                cache_i←cache_j+cache_i+limit;
        SynchronizeThreads();
        limit←limit/2;
likelihood_ij← likelihood_ij/cache0;
```

Algoritmo 4: CUDA mul-kernel paralelo

```
Input: DatabaseSize, DataSize, Samples, π_i, μ_i, ∑
Output: likelihoods
    i←ThreadIndex.X; //x, y, z
    j←BlockIndex.X;
    cache ← sharedmemory;// guarda os dados numa matriz 3D
    If sampleIdx < DatabaseSize;
        cache ← 0;
        If tidx < DataSize
            For 0 até DataSize do
                cache ← (sample - means)* S^-1; //direto da
memória global
            cache ← sum todas as componentes de cache2;
        SynchronizeThreads();
        While i != 0 // i = blockDim.x/2
            If tidx < i
                cache[index] += cache[index+i];
                SynchronizeThreads();
            i /= 2;
        If tidx == 0
                 likelihood ← cache;
```

III. METODOLOGIA

Sabe-se que para aumentar o nível de paralelização, devemos evitar a implementação de laços dentro dos kernels sempre que os cálculos forem linearmente independentes [10]. No caso dos algoritmos EM, temos kernels cujos resultados são independentes um do outro, permitindo a paralelização dos mesmos e, dentro de cada kernel, havia o uso de laços desnecessários [3]. Para haver melhor acesso à memória da GPU, é necessário conhecer seu tamanho e velocidade de acesso, fazendo um tradeoff de acordo com a construção do algoritmo [2], [10].

A. *Funcionamento e deficiências do Kernel Mul*

Na solução apresentada por [3], o kernel Mul utilizava 3 construções que foram analisadas e modificadas conforme a redução dos laços. O kernel Mul do GMM estava sendo alocado numa dimensão 2D em cada bloco e cada bloco realizava um laço que se repetia segundo o cálculo do tamanho da base de dados dividido pelo número de gaussianas vezes 4. Esta construção adicionava uma complexidade de O(N) para o algoritmo. Com a dimensão 2D, foi declarado um grid dim3 dimGrid(1, número de gaussianas, 1).

O primeiro laço, que iremos denominar de *while1*, realizava apenas uma cópia da matriz de covariância inversa para uma memória compartilhada matrix. O laço seguinte, *while2*, era dividido em duas etapas, onde a primeira fazia a subtração entre o vetor de amostras e o vetor de médias (x - u) e armazenava numa outra memória compartilhada, chamada *cache1*. A segunda fazia o cálculo da redução com o somatório da multiplicação entre matrix e *cache1* (x - u * S-1), armazenando seu resultado em *cache2*, e em seguida multiplicava *cache1* por *cache2*, novamente armazenando o resultado da operação em *cache2*.

B. *Proposta de melhorias no Kernel Mul*

Para otimizar este kernel, o laço *while1* foi completamente removido. O vetor de amostra e de média nunca modificam e já estão armazenados na memória. Por isso, é estranho copiar novamente para memória compartilhada toda vez que se deseja realizar o Mul, se ele não mudou. Com a tarefa de remoção do *while1*:

- A matriz de covariância inversa deixou de estar na memória compartilhada,
- Foi evitado um processo de cópia,
- Aumentou o tempo de acesso à mesma,
- Mas tornou o restante do código mais fácil de paralelizar e otimizar.

Ao remover o *while1*, foi implementada uma terceira dimensão na chamada do kernel Mul e os blocos se tornaram 3D. Considere o Mul não otimizado como uma matriz onde cada elemento executava um laço. Cada elemento da matriz é representado por um bloco do grid, ou seja, cada bloco executa um laço antes da otimização.

Na nova arquitetura otimizada, ele seria um cubo e cada elemento do cubo/bloco realiza um único passo do que o antigo *while2* realizava e cada sub-cubo na profundidade (direção z), irá realizar outro passo.

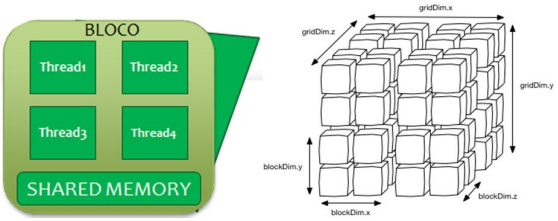

Figura 1. Bloco 2D em grid 3D.

Ao remover o *while1* é preciso atualizar os dados que antes pertenciam às memórias compartilhadas *matrix*, *cache1* e *cache2*. Foi criada apenas uma memória compartilhada chamada *cache*. Anteriormente, o kernel possuía 3 etapas:

1. Calcular (x - u) em *cache1*,
2. Calcular (x - u)*S-1 em *cache2*, fazendo o somatório de cada resultado e
3. Multiplicar *cache2* por *cache1*.

Após a otimização, o kernel realiza o cálculo 1 e 2 na mesma etapa e guarda em *cache*. Atualmente, cada cubo é um bloco do grid do CUDA e isso facilita a complexidade de entender e de escrever o código, uma vez que cada bloco tem sua própria memória compartilhada que poderia continuar sendo apenas 2D. O terceiro laço sofre mudança apenas no nome da memória compartilhada e continua executando a redução para somar a probabilidade desses dados.

Após alguns testes, notou-se que os kernels p e pn também poderiam ser otimizados usando o mesmo artifício de redução

de laços e redimensionamento da grid e blocos para se ter melhor acesso à memória.

IV. Testes, Resultados e Discussão

Foram realizados testes de execução com diferentes datasets, diferentes alocações de memória compartilhada e quantidades de gaussianas. Para haver comparação entre os tempos de execução dos kernels, foi realizado o teste do tempo de execução dos mesmos antes e depois da otimização variando-se de 1 até 1024 gaussianas. Em ambos os casos, para o valor de 1024 gaussianas ocorria a interrupção de funcionamento da placa de vídeo. Antes da otimização, para 512 gaussianas o tempo de processamento do kernel Mul chegava a ser maior do que o tempo da versão em CPU.

Em relação ao compartilhamento de memória, para obter melhor uso da mesma, foi necessário alocar memórias compartilhadas adicionais para cada bloco na direção Z (profundidade de um bloco no Grid). O tempo de alocação de memória é persistentemente maior do que o tempo de executar os códigos.

Com o compartilhamento de memória em cada bloco, os *threads* compartilham seu acesso. Assim, o tempo total de execução do kernel Mul foi reduzido significativamente, de 0.004s para 0.0001s. Acreditamos que o tempo atual de 0.004s é bom o suficiente se comparado com o obtido pela CPU, que é 0.290s.

A. *Computador utilizado nos testes*

Todos os experimentos foram realizados em um computador com CPU Intel(R) Core(TM) i5-3330 @3.00GHz (4CPUs), 8192MB RAM, GeForce GTX550ti 1GB DDR5.

B. *Datasets*

Cinco datasets de tamanhos diferentes foram utilizados. Cada um possui uma variação entre vozes de homens e mulheres, horas de gravação, frases, distância e tipo de microfone. Para cada um deles, foram extraídas as instâncias com 13 *Mel Frequency Cepstral Coeficientes* (MFCC), amplamente utilizado para representar os sinais de áudio em sistemas de processamento de fala, que normalmente usam GMM para modelar a distribuição de fonemas na linguagem [11]. As bases utilizadas foram:

- *Arabic Spoken Digit 9* do repositório da UCI [12], que consiste de 8800 instâncias (6600 para treinamento e 2200 para validação) correspondendo a áudios de 88 locutores (44 homens e 44 mulheres) pronunciando dígitos de 0 à 9;
- A base de dados *An4* [13] consiste de 948 instâncias de treinamento e 130 instâncias de teste. Cada locutor foi orientado a falar sobre informações pessoais, como nome, endereço, etc.
- A *CMULM Chaplain* [14] é uma base de diálogos de áudio com 4.15 horas de conversação.
- Do CMU PDA Database, escolhemos o dataset PDAm [15]. Neste dataset, as vozes foram gravadas por múltiplos microfones com um PDA. Consiste de 11 locutores, cada um realizando a leitura de 50 sentenças, resultando em 950 MB de arquivos de áudio WAV.
- A base de dados CMU SIN [10] contém 500 sentenças gravadas por um único locutor inglês do sexo masculino em um ambiente ruidoso.

C. *Variação da base de dados e número de gaussianas*

Realizando-se o teste para 8 gaussianas, o tempo de execução do kernel Mul não otimizado saiu de 0.075s para 0.004s e o tempo total do GMM em CUDA saiu de 0.078s para 0.008s. Assim, houve uma redução de 89,75% do tempo de execução no CUDA no passo E.

A princípio, buscava-se otimizar apenas o kernel Mul, uma vez que este possuía o maior tempo de execução do passo E. Com a otimização do kernel Mul obtendo o tempo de 0.004s, foi observado que o tempo do p_kernel, responsável por computar o vetor de probabilidades marginais de cada gaussiana do passo-E, era bem maior que o *Mul* (responsável por cálculos mais complexos), conforme aumentava-se o número de gaussianas.

A mesma técnica de paralelização foi então utilizada no p_kernel e pn_kernel. O pn_kernel teve uma melhora insignificante de 0.0001s, mas o p_kernel se manteve quase constante mesmo com a variação do número de gaussianas com um tempo de 0.00005s a 0.001s conforme o número de gaussianas cresce.

Para aumentar ainda mais o poder de paralelização dos kernels p e pn foi preciso tornar as operações independentes, o que não ocorria dentro do laço responsável pela soma dos valores da matriz de verossimilhança em pn_kernel.

As tabelas 1 e 2 mostram os tempos de execução em segundos para os 3 kernels do algoritmo EM conforme aumenta-se o número de gaussianas. Nota-se que os tempos estão praticamente constantes após a otimização na tabela 2.

Tabela 1. Tempo em segundos do kernel Mul, p e pn antes da otimização

Estimação Paralela na GPU			
nº gaussianas	***mul***	***p_kernel***	***pn_kernel***
8	0.07043	0.00017	0.00019
32	0.07412	0.00043	0.00066
64	0.08097	0.00085	0.00015
128	0.07912	0.00178	0.00234
256	0.08181	0.00339	0.00003
512	0.07565	0.00675	0.00003
1024	0.08049	0.01259	0.00004

Outro teste realizado foi fixar o número de gaussianas em 8 e aumentou-se a quantidade da base de dados variando entre 22300 e 223000. Esse experimento mostrou que conforme a base de dados cresce, o tempo de execução na CPU e GPU

aumenta, mantendo-se a proporção de tempo para cada um dos 5 diferentes datasets.

TABELA 2. TEMPO EM SEGUNDOS DO KERNEL MUL/P E PN DEPOIS DA OTIMIZAÇÃO DOS 3 KERNEIS.

Estimação Paralela na GPU			
nº gaussianas	***Mul***	***p_kernel***	***pn_kernel***
8	0.00419	0.00005	0.00015
32	0.00416	0.00006	0.00049
64	0.00418	0.00009	0.00095
128	0.00440	0.00016	0.00185
256	0.00424	0.00030	0.00003
512	0.00422	0.00058	0.00003
1024	0.00459	0.00112	0.00003

V. CONCLUSÃO

Este artigo tratou de projetar e executar procedimentos de paralelização do E-passo para o algoritmo EM de treinamento de GMM. Esta tarefa de otimização foi realizada a partir das observações de desempenho da solução apresentada por [3].

Após as otimizações providas, foi possível concluir que o número de gaussianas não possui interferência significativa para o tempo de execução dos kernels e conforme a base de dados cresce, o tempo de execução da CPU e GPU crescem de forma proporcional. A remoção de laços é essencial para implementar melhorias nos kernels paralelizados. Reduzir a complexidade dos algoritmos faz parte do processo de otimização que é independente da Ocupância, e isso é conseguido ao trocar os laços de repetição aninhados. O redimensionamento da arquitetura dos kernels para a terceira dimensão mostrou ser eficiente, aumentando a concorrência dos threads, garantindo melhor acesso à memória global e reduzindo significativamente o tempo de execução total.

Com o objetivo de identificar com maior propriedade outros gargalos nas rotinas paralelizadas e evoluirmos a otimização dos kernels implementados, estamos aplicando o método sistemático proposto em [16]. Em outro trabalho futuro, pretendemos aplicar a paralelização em GPU do algoritmo EM para treinamento de outros modelos.

AGRADECIMENTOS

Os autores agradecem ao Conselho Nacional de Desenvolvimento Científico e Tecnológico (CNPq) pelo suporte financeiro [Universal 14/2012, Processo 483437 / 2012-3].

REFERÊNCIAS

1. A. Yi and O. Talakoub. “Implementing a Speech Recognition System on a Graphics Processor Unit (GPU) using CUDA”. In: University of Toronto, 2009.
2. J. Sandres. “CUDA by example: an introduction to general-purpose GPU programming”. ISBN 978-0-13-138768-3 NVIDIA Corporation, 2011.
3. M. Medeiros et al. “Multi-kernel approach to parallelization of EM algorithm for GMM training”. In: IEE Conference, Brazilian Conference on Inteligent Systems (BRACIS14) p. 158-165 DOI: 10.1109/BRACIS.2014.38. São Paulo, October, 2014.
4. M. Azhari. “A CUDA based Parallel Implementation of Speaker Verification System”. In: Institute of Graduate Studies and Research in partial fulfillment of the requirements for the Degree of Master of Science; Eastern Mediterranean University, 2011.
5. H. Tagare, A. Barthel, F. Sigworth. An adaptive expectation-maximization algorithm with GPU implementation for electron cryomicroscopy. Journal of Structural Biology, v. 171, p. 256-265. DOI:10.1016/j.jsb.2010.06.004. 2010.
6. M. Azhari, C. Ergün. Fast Universal Background Model (UBM) training on GPUs using Compute Unified Device Architecture (CUDA). International Journal of Electrical & Computer Sciences IJECS-IJENS, Rawalpindi: v. 11, n. 4, p.49-55, ago. 2011.
7. C. Chen, D. Mu, H. Zhang, B. Hong. A GPU-accelerated approximate algorithm for incremental learning of gaussian mixture model. Proc. of IEEE 26th Int. Parallel and Distributed Processing Symposium, (WF‘12), Xangai: p. 1937-1943, mai. 2012.
8. N. Kumar, S. Satoor and I. Buck NVIDIA Corporation. “Fast Parallel Expectation Maximization for Gaussian Mixture Models on GPUs Using CUDA”. In: Proceedings of the 11th IEEE Int. Conf. on High Performance Computing and Communications (HPCC09), Houston: p. 103-109. 25-27, 2009.
9. L. Machlica, J. Vanek, Z. Zajic. Fast estimation of gaussian mixture model parameters on GPU Using CUDA. Proc. of the 12th Int. Conf. on Parallel and Distributed Computing, Applications and Technologies, (AT ‘11), Gwangju: p. 167-172, out. 2011
10. N. Kumar, S. Satoor and I. Buck. Fast parallel expectation maximization for gaussian mixture models on GPUs Using CUDA. Proc. of the 11th IEEE Int. Conf. on High Performance Computing and Commu., (CC‘09), Houston: p. 103- 109. 25-27 jun. 2009.
11. F. Zheng, G. Zhang and Z. Song. Comparison of different implementations of MFCC. Journal of Computer Science and Technology, v. 16, p. 582-589. 2001
12. K. Bache and M. Lichman. UCI Machine Learning Repository. Disponível em <http://archive.ics.uci.edu/ml>. Acesso em: 26 fev. 2014.
13. A. Acero. Acoustical and Evironmental Robustness in Automatic Speech Recognition. Tese (Doutorado em Engenharia Elétrica) – Department of Electrical and Computer Engeneering. Pittsburg: Carnegie Mellon University, Kluwer Academic Publishers Norwell, 1990.
14. A. Black. CMULM Chaplain: spoken dialog database Disponível em < http://www.speech.cs.cmu.edu/Tongues/ >. Acessado em: 26 fev 2014.
15. E. Gouvêa. The CMU Audio Databases. Disponível em <http://www.speech.cs.cmu.edu/databases/pda/ >. Acessado em: 26 fev 2014.
16. O. de Oliveira. MEPARALEL: um método para análise de implementação de algoritmo paralelo baseado em CUDA. Dissertação de Mestrado. Programa de Pós-Graduação em Ciência da Computação (PROCC). Universidade Federal de Sergipe, 2015.