Implementações do Algoritmo SLCS em CUDA e Aplicações em Recuperação de Informação Matemática

Alexandre W. Miya[1], **Guilherme C. Gomes**[1], **Flavio B. Gonzaga**[1]

[1]Departamento de Ciência da Computação – Universidade Federal de Alfenas (UNIFAL-MG)
CEP 37.133-840 – Alfenas – MG – Brasil

{a14004,a14038,fbgonzaga}@bcc.unifal-mg.edu.br

Resumo. *Como a web representa hoje nossa principal base de informações, é natural que cientistas e acadêmicos a utilizem como fonte de conteúdo matemático. Estudos mostram no entanto que, técnicas já populares utilizadas na busca e recuperação de conteúdo textual não geram bons resultados quando aplicadas à busca e recuperação de conteúdo matemático. Em vista disso, ao longo dos últimos anos, alguns estudos propuseram diferentes maneiras para se abordar o problema de maneira aceitável. Dentre as propostas, pode-se citar o uso de algoritmos de programação dinâmica. Assim, este estudo propõe o aproveitamento das propriedades das Unidades de Processamento Gráfico (GPU's), por meio da plataforma CUDA (Compute Unified Device Architecture), para a implementação de três abordagens paralelas ao problema de busca e recuperação de conteúdo matemático.*

1. Introdução

As comunidades acadêmica e científica pertinentes às áreas de ciências exatas têm o exercer de suas atividades fortemente atrelado à conceitos matemáticos. Assim, utilizam de forma recorrente uma enorme gama de fórmulas.

Dada a vasta quantidade de conteúdo *web* e a infinidade de fórmulas existentes, tais comunidades necessitam do auxílio de ferramentas de busca para viabilização de suas pesquisas na internet. Segundo [Kumar et al. 2012], a busca e recuperação de conteúdo matemático deve levar em conta o relacionamento e posição dos termos que compõem as expressões, diferentemente do que ocorre em abordagens tradicionais de recuperação textual que se baseiam na técnica *"bag of words"*, onde as expressões são tratadas como conjuntos de termos independentes e o contexto é desconsiderado. Portanto, existem atualmente poucos motores de pesquisa voltados especificamente para o domínio matemático.

Como os buscadores web exercem enorme quantia de trabalho, visto que operam em vastas bases de dados, é imprescindível que sua execução ocorra em sistemas de processamento paralelo. Usualmente o processamento é divido por meio de um sistema distribuído composto por vários servidores [Tadros 2015].

Dentre as técnicas que visam explorar o paralelismo de execução podemos destacar uma, denominada de GPGPU (*General Purpose Graphics Processing Unit*), pois vem demonstrando ser uma alternativa econômica e eficiente na resolução de problemas que apresentam grande paralelismo de dados. Tal técnica faz uso da grande quantidade de núcleos presentes em Unidades de Processamento Gráfico (ou GPU's, sigla do termo

em inglês: *"Graphics Processing Unit"*), permitindo sua utilização para o processamento de informações genéricas, que não precisam ter qualquer relação com computação gráfica [Owens et al. 2007].

A quantidade de plataformas que implementam a técnica GPGPU ainda é pequena. Em meio a estas plataformas, as mais notáveis são a OpenCL (*Open Computing Language*), a qual faz uso de um padrão de programação aberto que é elaborado pela organização sem fins lucrativos *Khronos Group*, e a CUDA (*Compute Unified Device Architecture*), que é uma tecnologia proprietária da NVIDIA [Fang et al. 2011], [Karimi et al. 2010].

Este artigo expõe o desempenho de três abordagens paralelas, utilizando a plataforma CUDA, para o problema da busca e recuperação de conteúdo matemático.

2. Referencial Teórico

Dado que a notação matemática se diferencia da representação da linguagem natural, pelo fato de possuir característica bidimensional, métodos convencionais de recuperação textual, baseados na técnica *"bag of words"*, não geram bons resultados quando aplicadas à busca de conteúdo matemático. Esta característica espacial desempenha papel fundamental na construção do significado de expressões matemáticas por carregar o que [Kumar et al. 2012] chama de informação estrutural. Nela o posicionamento dos termos expressa a relação existente entre eles. Assim, segundo [Zanibbi and Blostein 2012] as pesquisas em torno do problema de busca e recuperação de conteúdo na conjuntura matemática estão ainda em estágios iniciais quando comparadas ao âmbito textual.

Em vista disso, [Kumar et al. 2012] expõe uma abordagem que leva em conta as informações estruturais e a natureza bidimensional para aprimorar o processo. A proposta utiliza como base o algoritmo da subsequência comum mais longa (ou LCS, sigla do termo em inglês: *"Longest Common Subsequence"*), já que esse preserva a ordenação dos termos.

2.1. Subsequência Comum Mais Longa

Dada uma sequência X de caracteres, é possível obter uma subsequência por meio da omissão de alguns de seus elementos. Deste modo, uma subsequência contém somente elementos que estão presentes na respectiva sequência e a ordenação original de tais elementos é preservada. Uma subsequência é dita comum à duas sequências quando ela é, ao mesmo tempo, subsequência de ambas. Dadas duas sequências, o algoritmo LCS tem por objetivo encontrar a subsequência comum mais longa [Cormen et al. 2012].

O problema da LCS possui subestrutura ótima (pode ser subdividido e sua solução ótima contém soluções ótimas para os subproblemas), podendo ser resolvido de forma recursiva em tempo exponencial. No entanto, esta característica possibilita também a utilização do paradigma da programação dinâmica, que permite reduzir a complexidade de tempo para $\Theta(mn)$, onde **m** e **n** representam os comprimentos das sequências em questão [Cormen et al. 2012]. Tal redução de complexidade se deve à utilização de uma tabela para armazenamento dos resultados dos subproblemas e permite a resolução de forma iterativa, evitando a necessidade de recalcular valores de modo recursivo.

Por meio da programação dinâmica, o cálculo do comprimento da LCS utiliza uma tabela c[0..m,0..n] para armazenar os valores c[i,j] (obtidos por meio da Fórmula F1)

e uma tabela b[1..m,1..n] para auxiliar o processo de construção da subsequência. Ao se calcular c[i,j] é utilizada uma solução ótima de um subproblema (exceto quando i = 0 ou j = 0). Tal solução se baseia em um dos valores: c[i-1,j-1], c[i,j-1] ou c[i-1,j]. O critério de seleção do valor é também apresentado na Fórmula F1. Assim, b[i,j] deve apontar a origem da solução utilizada no cálculo de c[i,j]. O preenchimento das tabelas é iterativo e deve ocorrer da esquerda para a direita e de cima para baixo, ou seja, uma linha por vez, partindo-se do primeiro elemento à esquerda. Ao fim deste processo c[m,n] contém o tamanho de uma LCS e a subsequência em questão pode ser construída. A construção parte de b[m,n] e segue as direções apontadas: se b[i,j] = $\nwarrow$ e c[i,j] $\neq 0$, temos que $X_i = Y_j$, o respectivo elemento é um componente da LCS (é utilizada a operação de inserção na subsequência) e a construção deve prosseguir na direção indicada; se b[i,j] = $\leftarrow$, temos que $X_i \neq Y_j$, não há adição de elemento na LCS (é utilizada a operação de deleção em Y_j) e a construção deve prosseguir na direção indicada; se b[i,j] = $\uparrow$, temos que $X_i \neq Y_j$, não há adição de elemento na LCS (é utilizada a operação de deleção em X_i) e a construção deve prosseguir na direção indicada. Note que os elementos que compõem a LCS são identificados em ordem inversa [Cormen et al. 2012].

$$c[i,j] = \begin{cases} 0 & se\ i = 0\ ou\ j = 0, \\ c[i-1, j-1] + 1 & se\ i, j > 0\ e\ x_i = y_j, \\ max(c[i, j-1],\ c[i-1, j]) & se\ i, j > 0\ e\ x_i \neq y_j. \end{cases} \quad \text{(F1)}$$

[Cormen et al. 2012] ilustra o cálculo do comprimento da LCS, bem como o processo de identificação da subsequência (as tabelas c e b são agrupadas em uma única imagem para facilitar a compreensão), conforme pode ser verificado na Figura 1.

O algoritmo LCS pertence a uma família de técnicas que são utilizadas para análise de similaridade. Nela, a métrica de semelhança é construída em torno do conceito da distância de edição, que representa o número mínimo de operações (inserção, deleção, substituição e transposição) necessárias para que uma cadeia de caracteres se iguale a outra. [Wikipedia contributors 2018], destaca a distinção entre as técnicas de acordo com o conjunto de operações utilizadas:

- Subsequência Comum Mais Longa (LCS): utiliza operações de inserção e deleção;
- Distância de Hamming: permite somente a operação de substituição;
- Distância de Jaro: permite somente tranposições;
- Distância de Levenshtein: permite inserções, deleções e substituições;
- Distância de Damerau-Levenshtein: utiliza as operações de inserção, deleção, substituição e transposição.

2.2. *Structure Based* LCS

Conforme mencionado anteriormente, [Kumar et al. 2012] estende o algoritmo LCS para que a característica bidimensional das expressões matemáticas seja levada em consideração. Segundo ele, a incorporação de informações estruturais aprimora a relevância dos resultados da busca. Deste modo, a abordagem recebe o nome de *Structure based* LCS (SLCS).

Como a dependência existente entre os termos de uma expressão matemática é representada por meio das respectivas disposições espaciais e a técnica LCS utilizada como base já preserva a ordenação dos elementos, o conceito de informação estrutural se

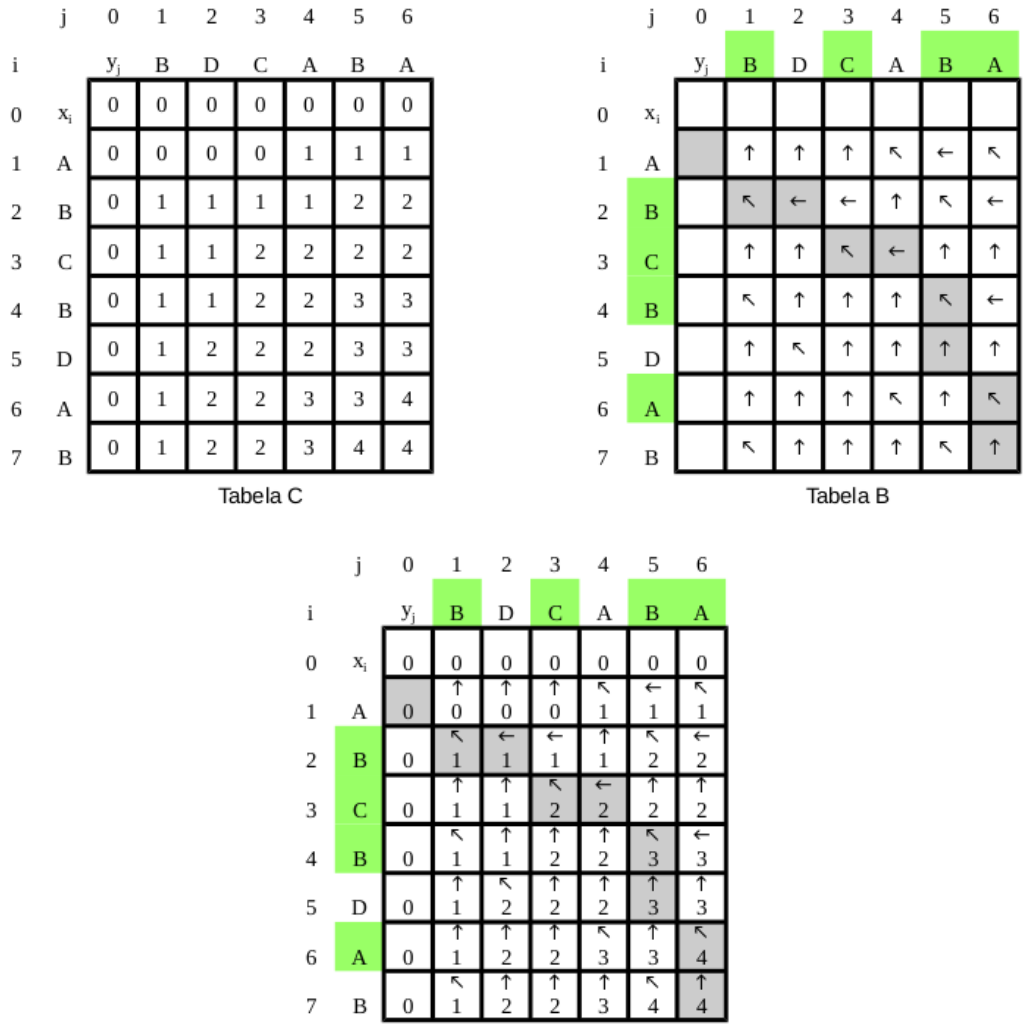

Figura 1. Cálculo das tabelas b e c sobre as sequências X = (A, B, C, B, D, A, B) e Y = (B, D, C, A, B, A). As células sombreadas representam o caminho percorrido no processo de construção da LCS. Os elementos destacados em verde são aqueles que compõem a LCS. c[7,6] contém o comprimento da LCS.

Fonte: baseado em [Cormen et al. 2012].

traduz no que [Kumar et al. 2012] chama de níveis. Estes níveis consistem em números inteiros que expressam o grau de aninhamento dos termos. Assim, para $\sqrt[n]{x_1}$ os níveis são apresentados na Tabela 1.

Termo	Nível
n	1
$\sqrt{}$	0
x	1
1	2

Tabela 1. Termos e os respetivos níveis

O acoplamento das informações estruturais ao LCS é obtido por meio da função

score, que retorna um valor real com base na diferença dos níveis referentes aos termos em questão. A função pode ser descrita pela seguinte fórmula:

$$score(Q[i], D[j]) = \frac{1}{|l(Q[i]) - l(D[j])| + 1} \quad \text{(F2)}$$

Nela, $Q[i]$ representa o i-ésimo elemento da expressão consultada (*query*), $D[j]$ representa o j-ésimo elemento de uma expressão oriunda da base de dados (*database*) e $l(x)$ representa o nível de x. Se os níveis de $Q[i]$ e $D[j]$ são diferentes, temos: $0 < score < 1$. Caso os níveis sejam iguais, o $score$ equivale a 1.

Em vista disso, o novo algoritmo pode ser expresso pela seguinte fórmula:

$$SLCS[i,j] = \begin{cases} 0 & se\ i = 0\ ou\ j = 0, \\ SLCS[i-1, j-1] + score(Q[i], D[j]) & se\ Q[i] = D[j], \\ max(SCLS[i, j-1],\ c[i-1, j]) & se\ Q[i] \neq D[j]. \end{cases} \quad \text{(F3)}$$

A ideia, portanto, do trabalho de [Kumar et al. 2012] é: dada uma *String* S_1 contendo uma consulta Q, e uma outra *String* S_2 contendo uma fórmula proveniente de uma base de dados D (ambas escritas em TEX), computar a similaridade entre S_1 e S_2, construindo uma matriz semelhante àquela mostrada na Figura 1, mas utilizando como regras as apresentadas na Fórmula F3. Antes que a similaridade possa ser computada no entanto, é realizada uma etapa de pré-processamento tanto em S_1 quanto em S_2, onde palavras-chave irrelevantes, como: \displaystyle, \begin{array}, etc, são eliminadas; números são trocados pela letra N; variáveis pela letra V; termos em sobrescrito e subscrito são marcados por P_s, P_e e B_s, B_e respectivamente, indicando o início e o término dos elementos em sobrescrito ou subscrito; e nomes de funções e palavras reservadas por códigos numéricos únicos. Por exemplo, \lim é trocado pelo código 225, \infty por 135, e assim por diante. Cada elemento desse ocupará uma posição na matriz na hora de fazer o cálculo de similaridade, conforme mostrado na Figura 2. Nela, $\lim_{x \to a} f(x) = L$ equivale a consulta Q e $\lim_{x \to \infty} f(x) = L$ é a fórmula proveniente da base de dados D. As respectivas representações após o pré-processamento são "255 B_s V 129 V B_e V (V) = V" e "255 B_s V 129 135 B_e V (V) = V", com níveis "0 1 1 1 1 1 0 0 0 0 0 0" e "0 1 1 1 1 1 0 0 0 0 0 0".

2.3. GPGPU

Em ferramentas de busca a quantidade de processamento envolvido na identificação de conteúdo relevante é proporcional ao tamanho da base de dados em questão. Considerando-se a vasta quantidade de conteúdo disponível na *web* e o fato de que, neste ambiente, a relevância dos motores de busca está relacionada também com a representatividade das respectivas bases de dados em relação à *web*, em geral, tais bases possuem tamanho considerável. Disto decorre que a quantidade de computação requerida tende a ser imensa. Daí a possibilidade de se explorar soluções baseadas em processamento paralelo.

Dentre as técnicas de processamento paralelo, uma tem se demonstrado muito promissora para problemas onde o paralelismo ocorre ao nível de dados, ou seja, quando há a necessidade de se submeter informações distintas a um mesmo conjunto de operações.

	j	0	1	2	3	4	5	6	7	8	9	10	11	12
i		y_j	255	B_s	V	129	135	B_e	V	(	V	)	=	V
0	x_i	0	0	0	0	0	0	0	0	0	0	0	0	0
1	255	0	1	1	1	1	1	1	1	1	1	1	1	1
2	B_s	0	1	2	2	2	2	2	2	2	2	2	2	2
3	V	0	1	2	3	3	3	3	2,5	2,5	2,5	2,5	2,5	2,5
4	129	0	1	2	3	4	4	4	4	4	4	4	4	4
5	V	0	1	2	3	4	4	4	4,5	4,5	4,5	4,5	4,5	4,5
6	B_e	0	1	2	3	4	4	5	5	5	5	5	5	5
7	V	0	1	2	2,5	4	4	5	6	6	6	6	6	6
8	(	0	1	2	2,5	4	4	5	6	7	7	7	7	7
9	V	0	1	2	2,5	4	4	5	6	7	8	8	8	8
10	)	0	1	2	2,5	4	4	5	6	7	8	9	9	9
11	=	0	1	2	2,5	4	4	5	6	7	8	9	10	10
12	V	0	1	2	2,5	4	4	5	6	7	8	9	10	11

Figura 2. Cálculo da tabela c sobre as sequências X = (255, B_s, V, 129, V, B_e, V, (, V,), =, V) e Y = (255, B_s, V, 129, 135, B_e, V, (, V,), =, V). c[12,12] contém o comprimento da SLCS.

Fonte: dos próprios autores.

Trata-se da GPGPU, técnica que permite tirar proveito do *hardware* de GPU's para realização de computação massiva sobre dados genéricos.

Em síntese, aplicações interativas que fazem uso de gráficos 3D têm como características marcantes a necessidade de grande quantidade de computação e a existência de paralelismo a nível de dados. Por esse motivo, GPU's são compostas por milhares de núcleos projetados e organizados de modo a trabalhar de forma paralela, porém uniforme, sobre uma vasta quantidade de dados distintos. Sua construção foca a capacidade de computação, em detrimento da complexidade das unidades de controle. Assim, possuem excelente relação custo/desempenho, que é o principal fator associado a atratividade da GPGPU.

Considerando-se a variedade de fabricantes e modelos de aceleradores gráficos, é natural que a implementação da GPGPU ocorra de diferentes formas. As plataformas mais difundidas são a OpenCL, a qual faz uso de um padrão de programação aberto, e a CUDA, que é uma tecnologia proprietária da NVIDIA. Segundo [Fang et al. 2011] e [Karimi et al. 2010] ambas as plataformas possuem performance comparável, com ligeira

e questionável superioridade da CUDA. A principal atratividade da OpenCL é sua portabilidade, aspecto que é irrelevante no âmbito deste projeto, dado que um ambiente para uma ferramenta de busca é previamente projetado, construído para essa finalidade, e não necessita de portabilidade. Assim, optou-se pela GPU da NVIDIA.

2.4. CUDA

A plataforma CUDA permite a manipulação de seus recursos por meio de um conjunto de instruções desenvolvido especificamente para a computação de propósito geral. A programação ocorre através de uma extensão das linguagens C/C++, que inclui um conjunto relativamente pequeno de novas palavras reservadas. Com isto, problemas genéricos não precisam ser disfarçados como aplicações gráficas e o desenvolvedor não necessita ter conhecimentos sobre a OpenGL (*Open Graphics Library*). Tais condições eram requisitos imprescindíveis nas primeiras abordagens de GPGPU [Sanders and Kandrot 2010].

Segundo [Chacón et al. 2014] e [Sanders and Kandrot 2010], o modelo de programação é baseado em uma hierarquia de *threads* que executam o mesmo programa sobre dados distintos. *Threads* podem ser agrupadas em blocos, de modo que as componentes de um mesmo bloco são capazes de cooperar utilizando registradores (*registers*) e memória *cache* compartilhada (*shared memory*).

Os blocos são alocados de forma não determinística para os multiprocessadores (*streaming multiprocessors* ou SM) da GPU, sendo que blocos distintos são executados de forma independente (podendo seguir fluxos de execução distintos), conforme a arquitetura MIMD (*"Multiple Instruction, Multiple Data"* - múltiplas instruções, múltiplos dados). Cada bloco é dividido em *warps* (conjuntos de 32 *threads*), que representam as menores unidades de trabalho alocável.

Várias *warps*, oriundas de um ou mais blocos, são alocadas dinamicamente para execução em um mesmo multiprocessador. As *threads* de uma mesma *warp* são distribuídas entre os processadores (*streaming processors* ou SP) de um multiprocessador. Estes processadores executam as mesmas operações, de modo sincronizado, sobre os dados (que se distinguem entre as *threads*), conforme a arquitetura SIMD (*"Single Instruction, Multiple Data"* - mesma instrução, múltiplos dados).

O conjunto de todos os blocos constitui uma estrutura chamada de grade (ou *kernel*), que representa a porção da aplicação a ser executada na GPU. O modelo de programação engloba também o conceito de *stream*, que representa uma fila de operações (chamadas de *kernel*, operações de memória e operações com eventos) da GPU onde a execução segue uma ordem específica, determinada pela sequência em que as operações são adicionadas.

Além dos registradores e da memória compartilhada, a arquitetura conta com dois tipos de memória *cache*: a *constant memory* e a *texture memory*. Ambas são responsáveis por armazenar dados que não serão modificados durante a execução do *kernel* (somente para leitura). A distinção entre elas reside no fato de que a segunda é utilizada para dados que possuem padrão de acesso específico, com localidade espacial, ou seja, diferentemente dos esquemas tradicionais de *cache* em CPU, onde são carregados endereços consecutivos, a *texture memory* considera a vizinhança num sentido bidimensional e carrega dados que não necessitam residir em endereços sequenciais. Há ainda a memória global (*global memory* ou *device memory*) que equivale à memória principal da GPU.

As Figuras 3 e 4 representam de forma sucinta a arquitetura cuda e facilitam a compreensão das informações transmitidas acima.

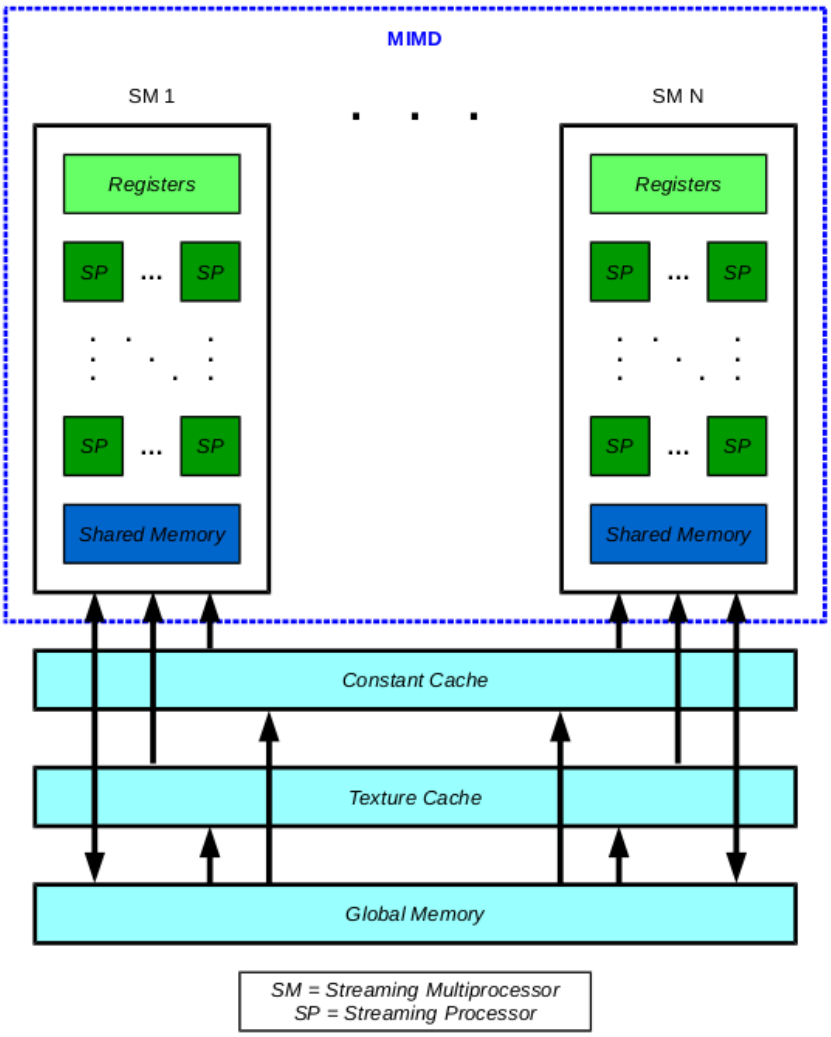

Figura 3. Arquitetura CUDA.
Fonte: dos próprios autores.

2.5. CUDA no Contexto da Similaridade Entre Cadeias de Caracteres

Em [Chacón et al. 2014] é proposta uma solução em CUDA para o problema de alinhamento de sequencias de DNA. Nela, utilizando-se a programação dinâmica, duas sequencias são analisadas para a localização de regiões similares e identificação de relações funcionais, estruturais ou evolucionárias entre elas. Para isso é utilizada a colaboração entre *threads* com dois níveis de paralelização: paralelização interna de uma tarefa (*intra-task parallelism*) - para aproveitamento da característica SIMD - e entre tarefas (*inter-task parallelism*) - para tirar proveito da característica MIMD.

Em [Balhaf et al. 2016] é exposta uma implementação paralela, em CUDA, para o cálculo da distância de Levenshtein (distância de edição). Nela, é utilizada uma técnica de rastreamento em diagonal, que permite reduzir a dependência de dados existente na matriz de programação dinâmica. Como o cálculo de uma célula da matriz está vinculado aos valores armazenados em outras três células, existe forte dependência de dados,

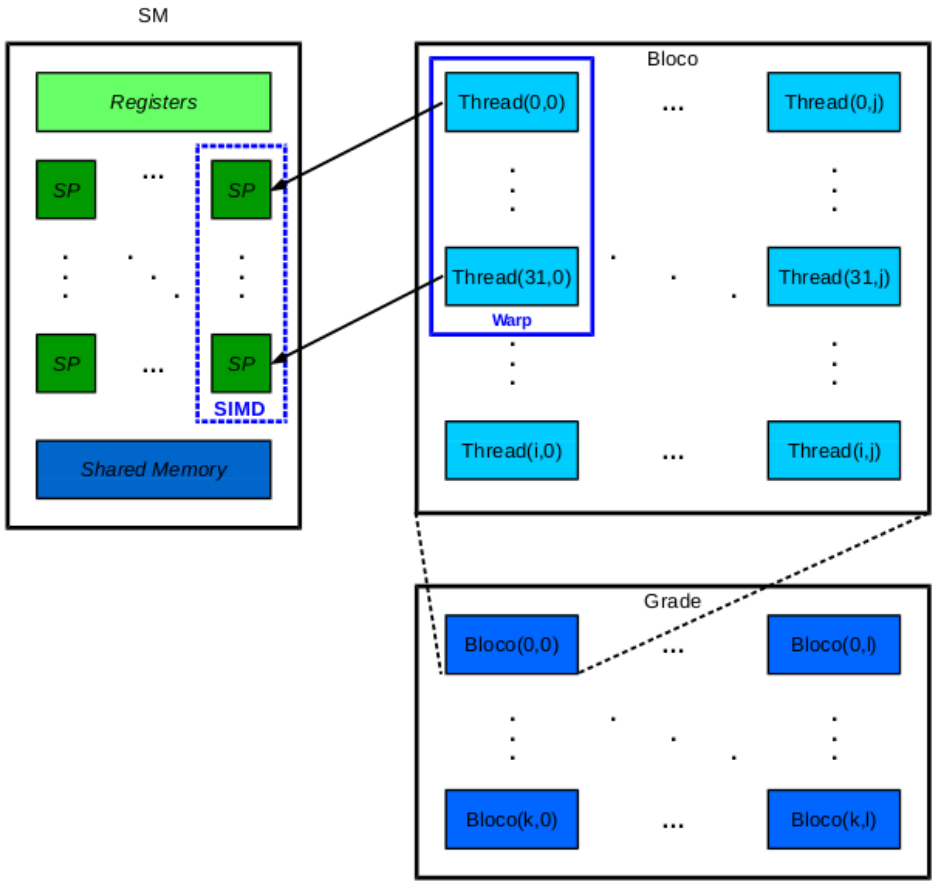

Figura 4. Representação dos conceitos CUDA.
Fonte: dos próprios autores.

conforme ilustrado na Figura 5. Deste modo, nas implementações convencionais, onde o preenchimento da matriz se dá linha por linha, apenas uma célula pode ser calculada por iteração. No entanto, a técnica proposta itera pelas diagonais, permitindo que em cada iteração todos os elementos da respectiva diagonal sejam executados em paralelo. Assim, a primeira iteração engloba somente o cálculo da célula [0,0]; a segunda iteração engloba o cálculo simultâneo das células [0,1] e [1,0]; as demais iterações seguem a mesma lógica. As Figuras 6 e 7 representam tal ideia.

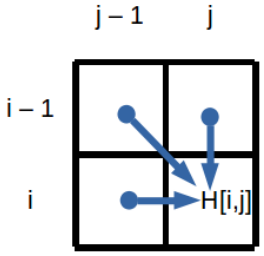

Figura 5. Problema da dependência de dados.
Fonte: baseado em [Balhaf et al. 2016].

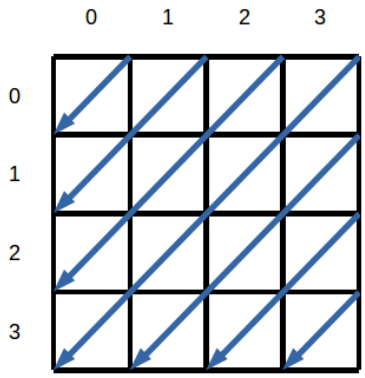

Figura 6. Técnica diagonal. Cada seta representa uma iteração
Fonte: baseado em [Balhaf et al. 2016].

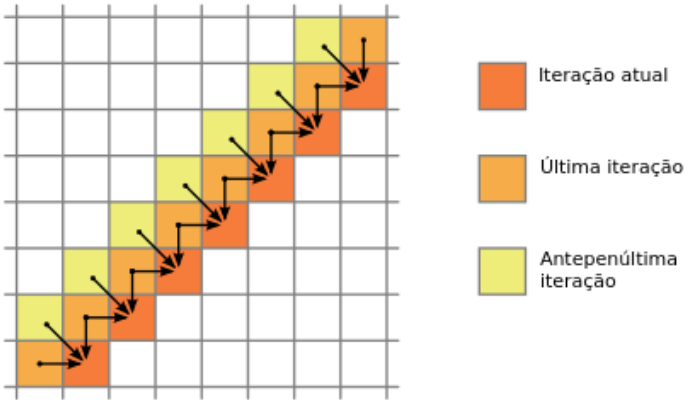

Figura 7. Dependências na matriz e as diagonais.
Fonte: baseado em [Bednárek et al. 2017].

3. Metodologia

Este trabalho propõe três abordagens que visam explorar diferentes níveis de paralelismo no processo de busca e recuperação de conteúdo matemático por meio da tecnologia CUDA:

Na primeira implementação, o foco é a paralelização interna de uma única tarefa, ou seja, dadas duas expressões matemáticas, o cálculo da subsequência comum mais longa é fragmentado para que seja possível explorar o paralelismo proveniente da subestrutura ótima do problema. Assim, são aproveitados os conceitos expostos em [Kumar et al. 2012] e [Balhaf et al. 2016], de modo que duas expressões matemáticas são submetidas ao algoritmo SLCS e a paralelização se dá por meio da técnica de rastreamento em diagonal: a cada iteração a CPU identifica quais elementos da tabela podem ser calculados em paralelo, de acordo com a respectiva diagonal, e efetua a chamada do *kernel* para execução dos cálculos na GPU. Para cada elemento haverá um bloco e em cada bloco a execução ocorrerá em uma única *thread*. A Figura 8 ilustra esta implementação durante a execução da terceira iteração.

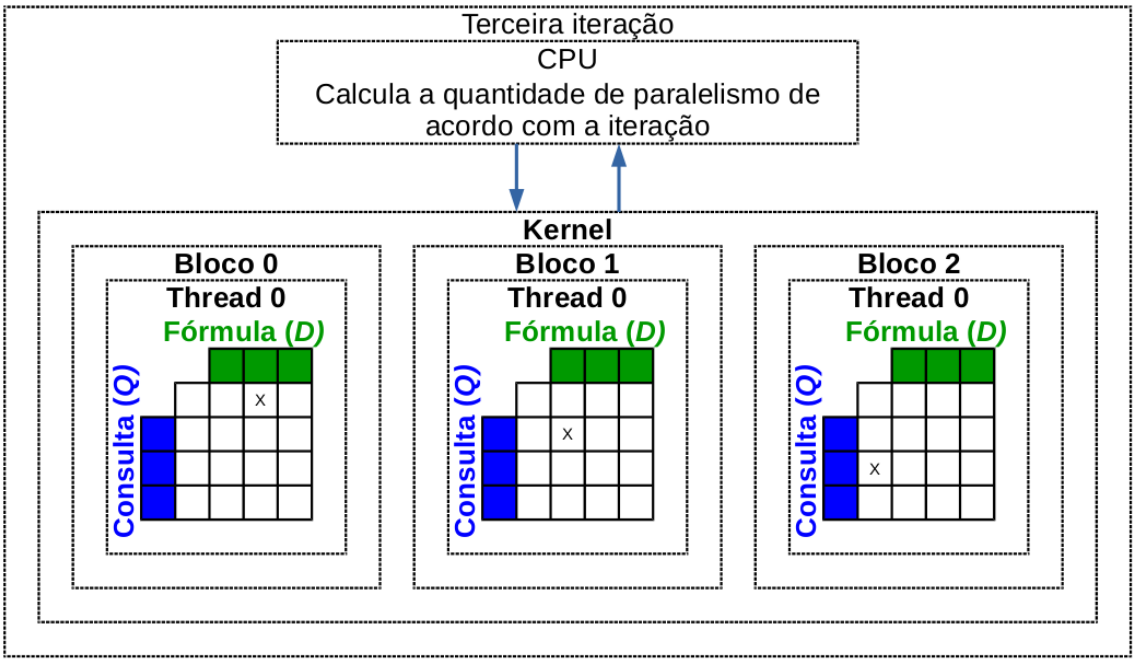

Figura 8. Ilustração da primeira implementação durante a execução da terceira iteração.

Fonte: dos próprios autores.

A segunda implementação se baseia no algoritmo SLCS de [Kumar et al. 2012] e tem por objetivo a paralelização ao nível de tarefas: a avaliação de similaridade entre a expressão consultada Q e cada fórmula D disponível no banco de dados ocorre de forma simultânea. Há, portanto, várias tarefas distintas, cada uma com o objetivo de calcular a respectiva subsequência comum mais longa. Tais tarefas são independentes e cada uma possui sua própria tabela [0..m,0..n] (onde n varia de acordo a fórmula D em questão). A implementação se baseia em uma única chamada de kernel onde cada bloco se encarrega de uma tarefa. Em cada bloco há apenas uma *thread*, que executa o algoritmo SLCS de forma sequencial. Os dados são representado por meio de um vetor de fórmulas que contém em seu início a consulta Q e na sequencia as fórmula provenientes da base de dados. A Figura 9 representa tais ideias.

A terceira implementação busca aumentar paralelismo. Consiste, portanto, na combinação das duas primeiras implementações e utiliza também dos conceitos expostos em [Kumar et al. 2012] e [Balhaf et al. 2016]. Assim, várias tarefas são executadas de modo simultâneo, e em cada uma delas a técnica de rastreamento em diagonal é aplicada. Para isso, é utilizado o conceito de paralelismo dinâmico, que possibilita o aninhamento de *kernel*. Deste modo, temos um *kernel* principal, chamado de pai, que terá um bloco para cada uma das k fórmulas oriundas da base de dados. Cada bloco possui apenas uma *thread*, que é responsável por identificar o grau de paralelismo disponível, conforme iteração, respectiva diagonal e fórmula em questão. Representamos esse grau por G_f, onde $f = 0, 1, ..., k$. Feito isso, cada *thread* invoca um *kernel* filho, responsável por executar o algoritmo SLCS. Cada um dos k filhos possui G_f blocos e cada um desses blocos apenas uma *thread*. Assim, como na segunda implementação, os dados são representados por meio de um vetor de fórmulas. A Figura 10 ilustra esta implementação.

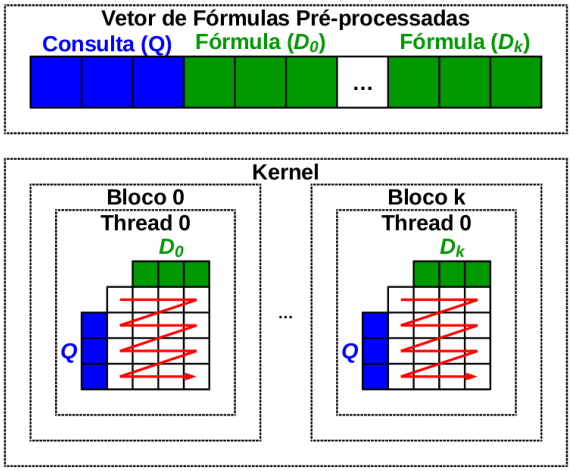

Figura 9. Ilustração da segunda implementação.
Fonte: dos próprios autores.

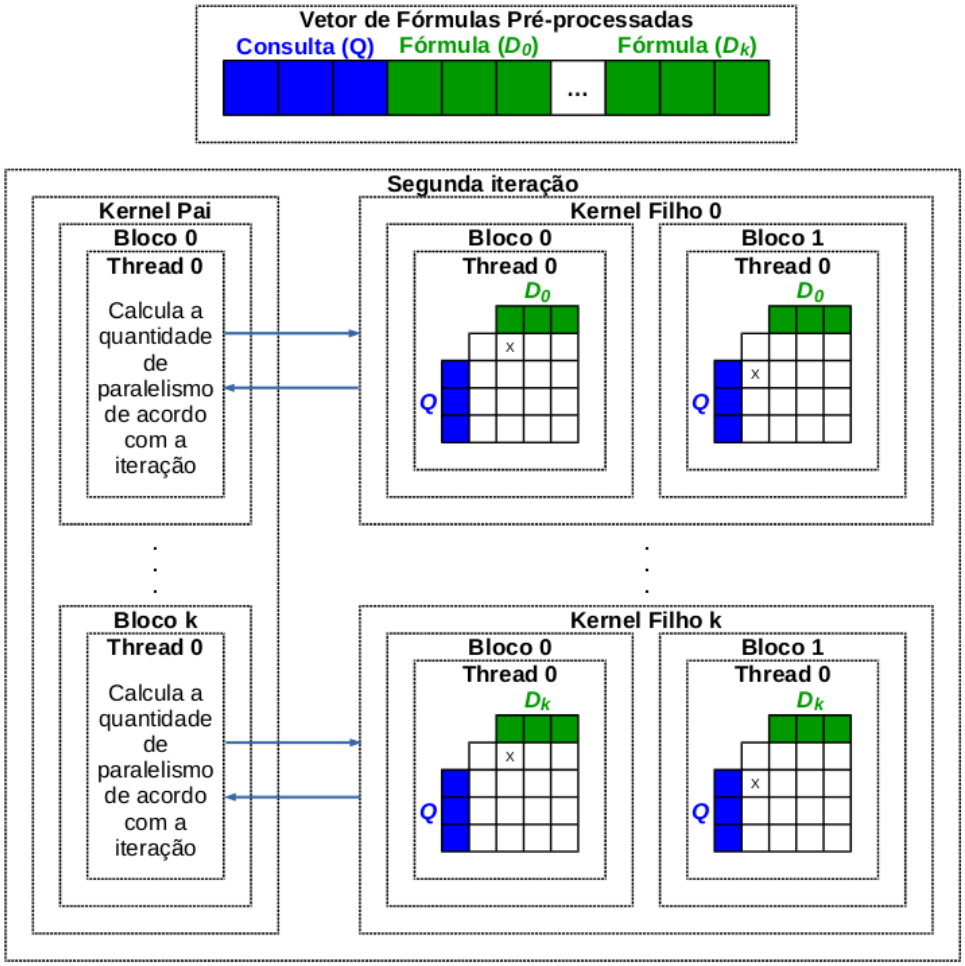

Figura 10. Ilustração da terceira implementação durante a execução da segunda iteração.

Fonte: dos próprios autores.

4. Resultados

Os experimentos foram realizados no seguinte sistema: GPU NVIDIA GTX 1050 Ti com *compute capability* 6.1, utilizando *driver* na versão 384.130 e CUDA *toolkit* V8.0.61; CPU Intel Core i7-7700 @ 3.6 GHz, de quatro núcleos físicos; 16 Gb de memória RAM; sistema operacional Ubuntu 16.04.4 LTS.

Foram efetuados testes em dois contextos distintos: um comparando a abordagem sequencial de [Kumar et al. 2012] e as três implementações descritas na metodologia quanto ao desempenho e escalabilidade em função do tamanho da base de dados; e outro analisando o comportamento da abordagem sequencial e de nossa primeira implementação em função do tamanho das fórmulas (quantidade de termos). No primeiro contexto, as bases de dados foram simuladas partindo-se de cinco expressões pré-processadas extraídas de [Kumar et al. 2012] e mostradas na Tabela 2. Essas expressões foram replicadas até se obter bases de tamanhos significativos. No segundo contexto, uma das expressões é selecionada e concatenada consigo mesma por várias vezes, até se obter uma expressão de comprimento significativo.

Notação TEX	Representação após pré-processamento	Níveis
$\lim_{x \to a} f(x) = L$	255 B_s V 129 V B_e V (V) = V	0 1 1 1 1 1 0 0 0 0 0 0
$\lim_{x \to \infty} f(x) = L$	255 B_s V 129 135 B_e V (V) = V	0 1 1 1 1 1 0 0 0 0 0 0
$\lim_{x \to a^+} f(x) = L$	255 B_s V 129 V P_s + P_e B_e V (V) = V	0 1 1 1 1 2 2 2 1 0 0 0 0 0 0
$\lim_{x \to a^-} f(x) = L$	255 B_s V 129 V P_s - P_e B_e V (V) = V	0 1 1 1 1 2 2 2 1 0 0 0 0 0 0
$\lim_{x \to -\infty} f(x) = L$	255 B_s V 129 - 135 B_e V (V) = V	0 1 1 1 1 1 1 0 0 0 0 0 0

Tabela 2. Fórmulas extraídas de [Kumar et al. 2012] e utilizadas na simulação das bases de dados.

As Figuras 11 e 12 apresentam os resultados. Os valores medidos consideram apenas os tempos de execução e retorno dos resultados, deixando de lado o carregamento de dados, pois se presume que, em uma aplicação real de busca e recuperação, as informações já estarão disponíveis para rápido acesso.

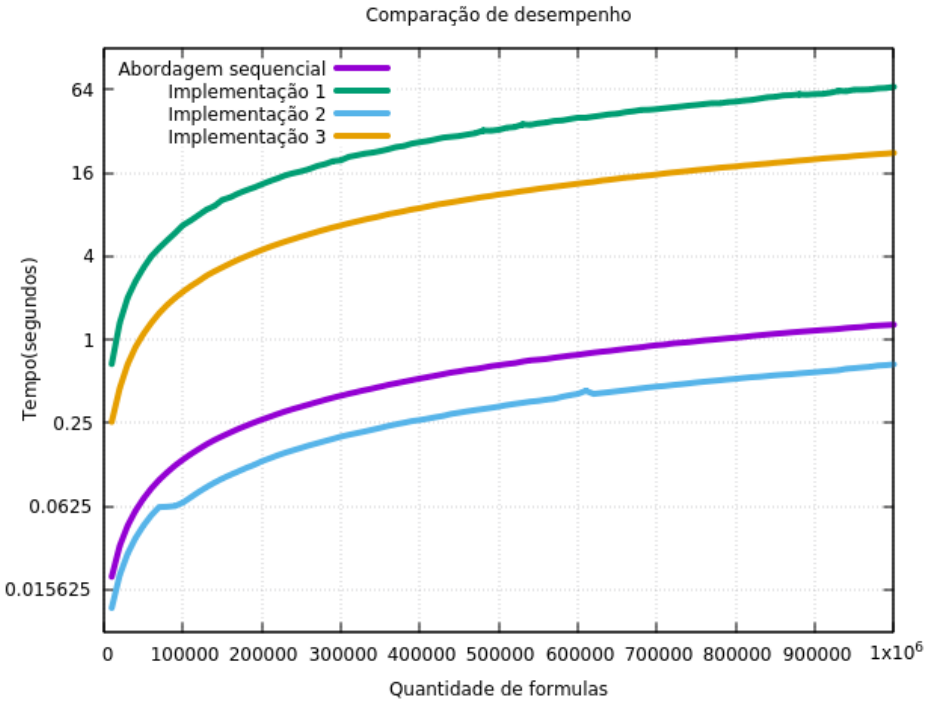

Figura 11. Desempenhos em função do tamanho da base de dados.
Fonte: dos próprios autores.

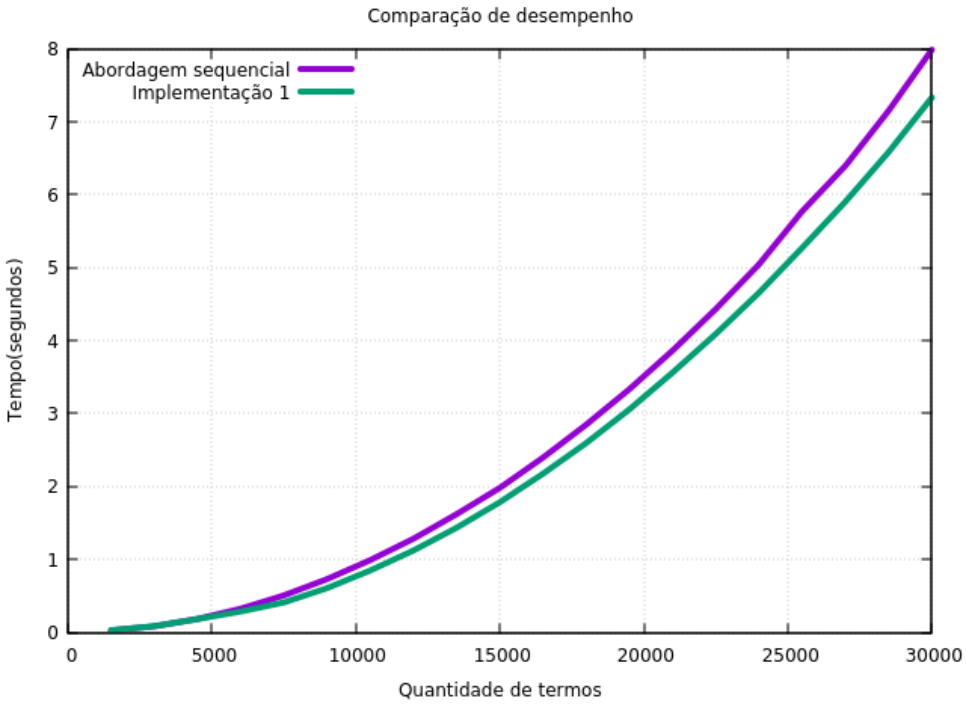

Figura 12. Desempenhos em função do tamanho das fórmulas (quantidade de termos).

Fonte: dos próprios autores.

5. Conclusão e Trabalhos Futuros

A plataforma CUDA é uma potente ferramenta de processamento massivo, permitindo a exploração de paralelismo por meio de abordagens diversas. Entretanto, sua aplicação se destaca em problemas onde há paralelismo ao nível de dados. A técnica LCS utilizada como base no trabalho de [Kumar et al. 2012] possui natureza recursiva e, por isso, apresenta considerável dependência de dados. A técnica de rastreamento em diagonal exposta em [Balhaf et al. 2016] busca contornar esse problema, no entanto, no contexto deste artigo, introduziu sobrecarga. Em vista disso, o melhor desempenho ocorreu na segunda implementação, onde somente o paralelismo de tarefas foi explorado.

Testes breves indicaram que, no contexto deste estudo, a distribuição de trabalho entre blocos e *threads* não interfere de forma significativa no desempenho. Em trabalhos futuros, experimentos apurados podem ser realizados em bases de dados diversificadas, visando enriquecer as conclusões em torno do assunto. Também podem ser analisadas formas de eliminar a sobrecarga citada, bem como implementações com a utilização de múltiplas *streams*, *texture memory* e *constant memory*.

Referências

[Balhaf et al. 2016] Balhaf, K., Shehab, M. A., Al-Sarayrah, W. T., Al-Ayyoub, M., Al-Saleh, M., and Jararweh, Y. (2016). Using gpus to speed-up levenshtein edit distance computation. In *2016 7th International Conference on Information and Communication Systems (ICICS)*, pages 80–84.

[Bednárek et al. 2017] Bednárek, D., Brabec, M., and Kruliš, M. (2017). Improving matrix-based dynamic programming on massively parallel accelerators. *Information Systems*, 64(Supplement C):175 – 193.

[Chacón et al. 2014] Chacón, A., Marco-Sola, S., Espinosa, A., Ribeca, P., and Moure, J. C. (2014). Thread-cooperative, bit-parallel computation of levenshtein distance on gpu. In *Proceedings of the 28th ACM International Conference on Supercomputing*, ICS '14, pages 103–112, New York, NY, USA. ACM.

[Cormen et al. 2012] Cormen, T., Leiserson, C., and Stein, R. (2012). *Algoritmos: teoria e prática*. ELSEVIER EDITORA.

[Fang et al. 2011] Fang, J., Varbanescu, A. L., and Sips, H. (2011). A comprehensive performance comparison of cuda and opencl. In *2011 International Conference on Parallel Processing*, pages 216–225.

[Karimi et al. 2010] Karimi, K., G. Dickson, N., and Hamze, F. (2010). A performance comparison of cuda and opencl. arXiv:1005.2581.

[Kumar et al. 2012] Kumar, P. P., Agarwal, A., and Bhagvati, C. (2012). A structure based approach for mathematical expression retrieval. In *MIWAI*, pages 23–34. Springer.

[Owens et al. 2007] Owens, J. D., Luebke, D., Govindaraju, N., Harris, M., Krüger, J., Lefohn, A. E., and Purcell, T. J. (2007). A survey of general-purpose computation on graphics hardware. *Computer Graphics Forum*, 26(1):80–113.

[Sanders and Kandrot 2010] Sanders, J. and Kandrot, E. (2010). *CUDA by Example: An Introduction to General-Purpose GPU Programming*. Addison-Wesley Professional, 1st edition.

[Tadros 2015] Tadros, R. (2015). *Accelerating web search using GPUs*. PhD thesis, University of British Columbia.

[Wikipedia contributors 2018] Wikipedia contributors (2018). Levenshtein distance — Wikipedia, the free encyclopedia. `https://en.wikipedia.org/w/index.php?title=Levenshtein_distance&oldid=846659618`. [Online; accessed 16-July-2018].

[Zanibbi and Blostein 2012] Zanibbi, R. and Blostein, D. (2012). Recognition and retrieval of mathematical expressions. *International Journal on Document Analysis and Recognition (IJDAR)*, 15(4):331–357.