String Matching Aproximado Utilizando Abordagem *Thread-Cooperative* em GPU do Algoritmo de Myers

Alexsander Andrade de Melo
Ygor de Mello Canalli

Universidade Federal Rural do Rio de Janeiro
Instituto Multidisciplinar

Nova Iguaçu - RJ, 04 de dezembro de 2014

Resumo

Este trabalho baseia-se no artigo *Thread-cooperative, Bit-parallel Computation of Levenshtein Distance on GPU* de Chacón et al. [1], no qual é apresentado duas estratégias de paralelismo em GPU para o problema de *String Matching* utilizando o algoritmo de Myers [5], a saber: uma estratégia inter-tarefa, ou seja, paralelismo a nível de tarefa (*Task-parallel*) e outra intra-tarefa (*Thread-cooperative*).

Neste traballho propomos uma estratégia *Thread-cooperative* diferente da abordada em [1], onde obtemos uma melhora de até 3 375% (33,75x) se comparado aos resultados obtidos na execução sequencial em CPU.

1 Introdução

Segundo Hyyrö [2], o problema de *string matching* aproximado pode ser definido, de maneira geral, como a busca por uma sub-strings em um texto basesado num limiar pré-definido de distância de edição em relação a um padrão dado. O *string matching* aproximado é um problema clássico em ciência da computação, com vastas aplicações, tais como: correção ortográfica, bioinformática e processamento de sinais [3].

Um dos métodos clássicos para realizar *string matching* é baseado no Algoritmo de Levenshtein [4], o qual possui uma complexidade de tempo e

espaço de $\mathcal{O}(n \times m)$, o que pode ser um grande obstáculo dependendo dos tamanhos de n e m. Com o ojbetivo de contribuir para o avanço das técnicas de *string matching*, sobretudo no que diz respeito ao desempenho, Myers [5] propôs uma alternativa para tratar o problema de *string matching* de maneira eficiente, mais especificamente através do uso de paralelismo a nível de bits.

Pesquisas recentes têm apresentado sucesso em utilizar placas gráficas para realizar computação de propósito geral (*General Prupose Graphical Processing Unit* - GPGPU) [6]. As placas gráficas geralmente possuem larga vatagem em relação à CPU para realizar tarefas que favoreçam o paradigma paralelo SIMD - *Single Instruction Multiple Data*, ou seja, instrução única, multiplos dados. E é nesta direção que o autor do artigo [1] aponta, se propondo melhorar ainda mais o desempenho do algoritmo de Myers através do uso de placas gráficas para realizar *string matching*.

1.1 Organização do trabalho

O trabalho é organizado como segue: na Seção 2 é apresentado o algoritmo tradicional de Levenshtein, para cálculo da *distância de edição*. Na Seção 2.2 é descrita a estratégia de paralelismo a nível de bits utilizada por Myers. Na Seção 3 são discutidas as estratégias de paralelismo abordadas pelo autor, bem como a estratégia adotada no desenvolvimento deste trabalho. Na Seção 4.1 são apresentadas as configuração utilizadas para realizar os experimentos aqui descritos. Na Seção 4.2 discutimos os resultados obtidos e realizamos uma breve análise de tais. Na Seção 5 falamos um pouco sobre os detalhes de otimização utilizados na implementação dos algoritmos. E, por fim, a Seção 6 sumariza as conclusões obtidas a partir deste trabalho, bem como apresenta possibilidades de trabalhos futuros.

2 Distância de Levenshtein

O algoritmo de Myers é na verdade uma modificação do algoritmo Levenshtein [4], cujo propósito é o cálculo da *distância de edição* entre duas strings a e b, isto é, o custo mínimo para transformar uma string a numa string b, considerando apenas as operações de inserção, remoção e substituição. Segue abaixo a definição do distância de Levenshtein utilizando relação de recorrência:

$$\text{lev}_{a,b}(i,j) \stackrel{\text{def}}{=} \begin{cases} \max(i,j) & \text{se } \min(i,j) = 0 \\ \min \begin{cases} \text{lev}_{a,b}(i-1,j)+1 \\ \text{lev}_{a,b}(i,j-1)+1 \\ \text{lev}_{a,b}(i-1,j-1)+\delta(i,j) \end{cases} & \text{caso contrário,} \end{cases} \tag{2.1}$$

onde $\text{lev}_{a,b}(i,j)$ denota a distância entre o prefixo de tamanho i da string a e o prefixo de tamanho j da string b e

$$\delta(i,j) \stackrel{\text{def}}{=} \begin{cases} 1, & \text{se } a[i-1] \neq b[i-1] \\ 0 & \text{caso contrário,} \end{cases}$$

$0 \leq i \leq n$ e $0 \leq j \leq m$, onde $|a| = n$ e $|b| = m$. Logo, o custo da distância de edição entre as strings a e b é dado por $\text{lev}_{a,b}(n,m)$.

Note que, como $\text{lev}_{a,b}(i,j)$ é o custo mínimo para transformar a string a na string b,

- $\text{lev}_{a,b}(i-1,j)+1$ refere-se ao custo de inserção do símbolo $a[i]$ ao se alinhar o prefixo de tamanho i de a com prefixo de tamanho j de b
- $\text{lev}_{a,b}(i,j-1)+1$ refere-se ao custo de inserção do símbolo $b[j]$ ao se alinhar o prefixo de tamanho i de a com prefixo de tamanho j de b
- $\text{lev}_{a,b}(i-1,j-1)+1$ refere-se ao custo de substituição do símbolo $a[i]$ pelo símbolo $b[j]$ ao se alinhar o prefixo de tamanho i de a com prefixo de tamanho j de b.

Tomemos como exemplo a Tabela 2.1 que ilustra o cálculo da distância de edição entre as strings $a =$ `'kitten'` e $b =$ `'sitting'` utilizando a distância de Levenshtein.

		k	**i**	**t**	**t**	**e**	**n**
	0	1	2	3	4	5	6
s	1	1	2	3	4	5	6
i	2	2	1	2	3	4	5
t	3	3	2	1	2	3	4
t	4	4	3	2	1	2	3
i	5	5	4	3	2	2	3
n	6	6	5	4	3	3	2
g	7	7	6	5	4	4	**3**

Tabela 2.1: Custo de edição entre as palavras `'kitten'` e `'sitting'` (pela distância de Levenshtein) é 3.

Observe que o algoritmo de Levenshtein pode ser calculado utilizando programação dinâmica com complexidade $\mathcal{O}((n+1)(m+1))$ de tempo e espaço, para tanto devemos preencher uma matriz $C_{n+1\times m+1}$ tal que

$$C_{i,j} = \text{lev}_{a,b}(i,j), \tag{2.2}$$

para $0 \leq i \leq n$ e $0 \leq j \leq m$. (Na verdade, há uma adaptação do algoritmo de Levenshtein para utilizar apenas $\mathcal{O}(n+1)$ de espaço, já que para calcular o valor de cada célula da matriz C necessitamos apenas dos valores da coluna imediatamente anterior à coluna corrente.) Ademais, é fácil notar que a matriz C pode ser preenchida de diversas formas, as mais comuns são: fixar a linha e variar as colunas, ou fixar a coluna e variar as linhas.

2.1 Utilizando o algoritmo de Levenshtein para o problema de *string matching* aproximado

Para utilizarmos a distância de Levenshtein para o problema de *String matching* devemos realizar uma pequena mudança na relação de congruência (2.1), a saber

$$\begin{array}{rcl} \text{lev}_{a,b}(i,0) & = & i, \\ \text{lev}_{a,b}(0,j) & = & 0, \end{array} \tag{2.3}$$

isso se deve ao fato que para verificarmos a ocorrência de um *matching* de a em b, não precisamos considerar o custo que antecede a ocorrência de $a[0]$ (ou seja, o primeiro símbolo de a) em b.

Nos problemas de *string matching* é comum chamar a string a de *padrão* e a string b de *texto*. Desta forma, de agora em diante, por questões de convenção de notação, assim o faremos, nomearemos a string a de *padrão* e a string t de texto, e denotaremos a e b por p e t, respectivamente. Além disso, chamaremos o valor $C_{n,j}$ de *score* de p no prefixo de tamanho j de t, e denotaremos por score_j, isto é,

$$\text{score}_0 = n$$

$$\text{score}_j \stackrel{\text{def}}{=} C_{n,j},$$

para $0 < j \leq m$.

Como estamos abordando o problema de *string matching aproximado*, estamos interessados em descobrir todas as ocorrências de j tais que

$$\text{score}_j \leq k, \tag{2.4}$$

para algum $k \geq 0$ dado, isto é, estamos interessados em localizar todas as ocorrências do padrão p que possui no máximo $k \geq 0$ diferenças com relação a uma fatia $t_z t_{z+1} \ldots t_j$ do texto t, onde $t = t_0 t_1 \ldots t_{m-1}$ e $0 \leq z \leq j < m$.

Observe que, se $k = 0$, estaremos tratando exatamente o problema de *string matching exato*. Observe ainda que, $n \leq m$ e, geralmente, $n \ll m$.

2.2 Algoritmo de Myers

Como dissemos anteriormente, o algoritmo de Myers é na verdade uma variação do algoritmo de Levenshtein, podendo ser aplicado tanto ao problema do cálculo da distância de edição quanto ao problema de *string matching*.

O algoritmo de Myers utiliza o bit-paralelismo para calcular a distância de Levenshtein. Para tanto, ele se utiliza da percepção devida ao Ukkonen [7] que a variação horizontal e a variação vertical entre cada célula da matriz de programação dinâmica C do Levenshtein se limita aos valores $\{-1, 0, +1\}$, ou seja, varia-se no máximo (no mínimo, respectivamente) ± 1, e a variação diagonal se limita aos valores $\{0, +1\}$. Com isso, foi proposto calcular a distância de Levenshtein utilizando variáveis binárias que representem tais variações, a saber as variáveis

$$\begin{array}{rcrcl} VP & \stackrel{\text{def}}{=} & \Delta v_{i,j} & == & +1 \\ VN & \stackrel{\text{def}}{=} & \Delta v_{i,j} & == & -1 \\ HP & \stackrel{\text{def}}{=} & \Delta h_{i,j} & == & +1 \\ HN & \stackrel{\text{def}}{=} & \Delta h_{i,j} & == & -1 \\ HX & \stackrel{\text{def}}{=} & \Delta d_{i,j} & == & 0, \end{array} \tag{2.5}$$

onde

$$\begin{array}{rcll} \Delta v_{i,j} & \stackrel{\text{def}}{=} & C_{i,j} - C_{i-1,j} & \text{variação vertical} \\ \Delta h_{i,j} & \stackrel{\text{def}}{=} & C_{i,j} - C_{i,j-1} & \text{variação horizontal} \\ \Delta d_{i,j} & \stackrel{\text{def}}{=} & C_{i,j} - C_{i-1,j-1} & \text{variação diagonal.} \end{array} \tag{2.6}$$

É possível determinar uma equivalência entre tais variáveis binárias de forma que, fixada uma coluna j, cada variável possa ser calculada mesmo sem conhecer os valores das células da matriz C. Para tanto, iremos tomar a variação horizontal como base para o cálculo do *score* de p e t. Desta forma, temos que

$$\text{score}_j = \text{score}_{j-1} + \Delta h_{m,j}. \tag{2.7}$$

Segue abaixo tais equivalências:

$$
\begin{array}{lcl}
HN_{i,j} & \Leftrightarrow & VP_{i,j-1} \text{ AND } HX_{i,j} \\
VN_{i,j} & \Leftrightarrow & HP_{i-1,j} \text{ AND } HX_{i,j} \\
HP_{i,j} & \Leftrightarrow & VN_{i,j-1} \text{ OR NOT } (VP_{i,j-1} \text{ OR } HX_{i,j}) \\
VP_{i,j} & \Leftrightarrow & HN_{i-1,j} \text{ OR NOT } (HP_{i-1,j} \text{ OR } HX_{i,j}) \\
HX_{i,j} & \Leftrightarrow & (VX_{i,j} \text{ AND } VP_{i,j-1}) + VP_{i,j-1}) \text{ XOR } VP_{i,j-1})) \text{ OR } EQ_{i,j}
\end{array}
\tag{2.8}
$$

onde $EQ_{i,j} \stackrel{\text{def}}{=} (p[i] == t[j])$ e $VX_{i,j} \stackrel{\text{def}}{=} EQ$ OR $VN_{i,j-1}$, para $0 \leq i \leq n$ e $0 \leq j \leq m$.

Note que, ao inicializarmos

$$VP_{i,0} = 1, \forall\, i = 0, 1, 2, \ldots, n$$

e ao inicializarmos

$$VN_{i,0} = 0, \forall\, i = 0, 1, 2, \ldots, n,$$

não interferimos diretamente no cálculo das variáveis (2.5) segundo às equivalências (2.8), e ao mesmo tempo viabilizamos o cálculo de tais.

Além disso, pela equação (2.7), necessitamos apenas calcular os valores de $HP_{i,j}$ e $HN_{i,j}$, que por sua vez precisam dos valores $VN_{i,j}$, $VP_{i,j}$ e $HX_{i,j}$. E, como os valores de $VN_{i,j}$, $VP_{i,j}$ são obtidos na iteração da coluna $j-1$ ou são inicializados, o cálculo do score_j pode ser facilmente realizado, não havendo portanto depedências circulares entre as variáveis binárias (2.5).

Uma observação importante (que é a grande vantagem do algoritmo de Myers) é a percepção que se $n \leq w$, onde w é o tamanho da palavra da unidade de processamento (geralmente $w = 32$ ou $w = 64$), e fixado um j, o conjunto de variáveis $HN_{i,j}$ para todo $i = 0, 1, \ldots, n$ pode ser representado como uma única variável, já que $HN_{i,j} \in \{0, 1\}$. Essa observação é aplicável igualmente para todas as variáveis binárias (2.5).

Com isso, ao se considerar $n \leq w$, o tempo de cálculo de todas as células de uma coluna da matriz C pode ser feito implicitamente em tempo $\mathcal{O}(1)$, através de um número constante de operações binárias. Logo, este algoritmo possui complexidade de tempo $\mathcal{O}(m)$ quando $n \leq w$. Ademais, Myers propôs também uma modificação neste seu algoritmo para incluir os casos em que $n > w$, obtendo então um algoritmo para o caso geral cuja a complexidade de tempo é $O(m[n/w])$. (Na verdade, para a primeira execução de um padrão, a complexidade do algoritmo de Myers tem complexidade $\mathcal{O}(n+m)$ quando $n \leq w$, pois é requerida uma etapa de pré-processamento, que será descrita em breve; da mesma forma, para os casos que incluem $n > w$, a complexidade do algoritmo de Myers é $O(m[n/w] + n + [n/w])$ para a primeira execução de um padrão.)

Com o intuito de permitir que todos os valores de $EQ_{i,j}$ para todo $i = 0, 1, \ldots, n$ sejam obtidos eficientementes (em tempo $\mathcal{O}(1)$) durante a execução do algoritmo de Myers, é realizada uma etapa de pré-processamento antes da execução do algoritmo (propriamente dito), que consiste em preencher uma matriz $PEq_{n\times|\Sigma|}$, onde $|\Sigma|$ é a quantidade de caracteres representados pelo esquema de codificação Σ utilizado (por exemplo, o ASCII possui tamanho 256), onde $p[i], t[j] \in \Sigma$, para $0 \leq i < n$, $0 \leq j < m$. Os valores da matriz $PEq_{n\times|\Sigma|}$ devem ser preenchidos da seguinte forma:

$$PEq[i,j] \stackrel{\text{def}}{=} \begin{cases} 1, & \text{se } p[i] == t[j], \\ 0, & \text{caso contrário.} \end{cases} \tag{2.9}$$

Assim, como $PEq[i,j] \in \{0,1\}$, e supondo que $n \leq w$, podemos representar a matriz $PEq_{n\times|\Sigma|}$ como um único vetor $PEq'_{|\Sigma|}$, onde todas as linha i de uma coluna j são representados por uma única variável. Já para o caso em que se permite $n > w$, a matriz $PEq_{n\times|\Sigma|}$ pode ser representada por uma matriz menor $PEq'_{\lceil n/w\rceil\times|\Sigma|}$. A complexidade de tempo para o preenchimento da matriz PEq' é $\mathcal{O}(n + \lceil n/w\rceil)$, e a complexidade de espaço é $\mathcal{O}(\lceil n/w\rceil \times |\Sigma|)$, que é o limite superior de espaço utilizado pelo algoritmo de Myers.

O algoritmo de Myers é descrito em Algoritmo 1, para os casos em que $m \leq w$.

3 Estratégia de paralelismo

No trabalho de Chacón et al [1] são utilizadas duas estratégias de paralismo sobre o algoritmo de Myers. A primeira é o *Task-parallel* onde o objetivo é paralelizar o cálculo do *score* de diferentes padrões para o mesmo texto. E a outra abordagem é a *Thread-cooperative*, na qual consiste em paralelizar o cálculo do *score* de um único padrão (com $n > w$) para um mesmo texto, onde cada *bloco do padrão* (ou seja, *substrings* de t com tamanho menor ou igual à w). Além disso, são apresentados resultados obtidos pela execução de ambas as estratégias juntas: *Task-parallel* e *Thread-cooperative*.

Neste nosso trabalho, não iremos realizar as mesmas abordagens feitas em [1], já que geralmente $n << m$, e há uma alta dependência de dados entre os blocos de um padrão, por conta das propagações geradas pelas operações de soma e *shift left*, o que prejudica o paralelismo, ainda mais a nível SIMD, que é o caso do CUDA, embora em [1] se tenha obtido bons resultados.

Nossa proposta consiste em uma abordagem *Thread-cooperative* cujo o objetivo é paralelizar o cálculo do *score* de um único padrão p num único texto de entrada t, o qual será dividido em fatias. O *score* de cada fatia do texto será computado de forma que o cálculo deste seja totalmente independete

Algoritmo 1 Algoritmo de Myers

Entrada: padrão p, texto t, $|p| = n$, $|t| = m$, esquema de codificação Σ

```
1: função MYERS(p, n, t,m, Σ)
2:     PEq' = PREPROCESSINGPEQ(p, n, Σ)
3:     para j = 0 até m faça
4:         EQ = PEq'[t[i]]
5:         VP = 2^m − 1
6:         VN = 0
7:         score = n
8:         score+ =ADVANCEDBLOCK(EQ, t, n, Σ)
9:         se score ≤ k então
10:            report occurrence at position j
11:        fim se
12:    fim para
13: fim função

14: função PREPROCESSINGPEQ(p, n, Σ)
15:    para i = 0 até |Σ| faça
16:        PEq'[i] = 0
17:    fim para
18:    para i = 0 até n faça
19:        PEq[p[i]] = PEq[p[i]] or (1 << i)
20:    fim para
21:    retorne PEq'
22: fim função

23: função ADVANCEDBLOCK(EQ, t, m, Σ)
24:    VX = EQ or VN
25:    HX = (((EQ and VP) + VP) xor VP) or EQ
26:    HP = VN or not (VP or HX)
27:    HN = VP and HX
28:    HP = HP << 1
29:    HN = HN << 1
30:    VP = HN or not (HP or VX)
31:    VN = HP and VX
32: fim função
```

do cálculo dos *scores* das demais fatias. Cada thread ficará responsável pelo cálculo de um único *score*.

Com esta abordagem de paralelismo, ficamos sujeitos ao problema de *bordas* de *string-matching*, que é o problema ocasionado por dividir o texto em fatias, onde ocorrências do padrão próximas das bordas de duas fatias do texto (uma borda de fim e outra de início) não são dectadas. Para resolver este problema, cada *thread*, exceto a *thread* de maior id (ou seja, a última *thread*), ficará ecanrregada por, além de calcular o *score* de sua respectiva fatia do texto, calcular o *score* de uma área de sobreposição de tamanho $n-1$ da fatia seguinte, onde n é o tamanho do padrão.

Como um *warp* pode executar até 32 *threads* por vez, é interessante que a quantidade de *thread*s lançadas sejam sempre potências de dois. Em nossa abordagem cada *thread* será responsável pelo cálculo do *score* de uma única fatia, sendo cada *fatia* calculada uma única vez. Assim, temos que a quantidade de *threads* lançadas deve ser igual à quantidade fatias do texto. Segue abaixo, como calculamos a quantidade de fatias, e também a quantidade de *threads*:

$$n\_slices\_base = \lfloor m/n \rfloor. \tag{3.1}$$

Observe que a quantidade de fatias retornada pela equação (3.1) não necessariamente é uma potência de dois. Para obtermos tal sempre, iremos adotar a seguinte regra: a quantidade de fatias do texto é definida como sendo a maior potência de dois menor ou igual à n_slices_base, isto é,

$$n\_slices \stackrel{\text{def}}{=} 2^q \tag{3.2}$$

tal que $2^q \leq \lfloor m/n \rfloor = n\_slices\_base < 2^{q+1}$, para algum $q \in \mathbb{N}$. Cada fatia será composta pela seu tamanho base *slice_base* e por uma área de sobreposição *overlapping*. O tamanho base de uma fatia é definino como segue:

$$slice\_base \stackrel{\text{def}}{=} \lfloor m/n\_slices \rfloor. \tag{3.3}$$

No entanto, é possível que $m \bmod n \neq 0$, e nestes casos é desejável que os $m \bmod n$ símbolos "extras"sejam propagados pelo maior número possível de *threads*, para que se evite ao máximo sobrecarga de um pequeno grupo de *threads*. Para tanto, adotaremos o seguinte esquema de divisão de cargas referentes ao resto:

$$cooperation\_remaining \stackrel{\text{def}}{=} thread.id \geq (n\_slices - m \bmod n). \tag{3.4}$$

Ou seja, se $cooperation\_remaining = 1$, então o tamanho da fatia da respectiva *thread* será acrescido de um, caso contrário nada será alterado.

Além do tamanho de base e do acréscimo do resto, como dissemos anteriormente, cada fatia possui uma área de sobreposição cujo o tamanho é dado como segue abaixo:

$$overlapping\_length \stackrel{\text{def}}{=} n - 1. \tag{3.5}$$

(Observe que, $overlapping\_length < n \leq slice\_base$, logo nunca teremos uma área de sobreposição maior que a fatia, o que nos garante que não haverá acesso fora dos limites de memória se utilizados os índices corretos.)

Portanto, o tamanho total de cada fatia j é dado por

$$legth_j \stackrel{\text{def}}{=} = slice\_base_j + cooperation\_remaining_j + overlapping\_length_j, \tag{3.6}$$

o que está diretamente relacionado à carga de trabalho da *thread* responsável pelo cálculo da fatia j.

4 Resultados obtidos

Nesta seção apresentamos os resultados obtidos no uso da nossa estratégia de paralelismo em GPU, como também as configurações do experimentos. Por questões de simplicidade, restringimos os nossos testes apenas para os casos em que $n \leq w$.

4.1 Configuração dos experimentos

Nossos experimentos foram realizados em um computador utilizando o Linux Kernel 3.13.0-40, com as seguintes configurações:

- Processador: Intel Core i5 4200U, dual core, 1.6 GHz
- 4GB de memória RAM
- Placa gráfica: Nvidia GT 740M, 2 multiprocessadores (sms) com 192 CUDA core cada (384 CUDA cores totais), 1.03 GHz, 64bits
- CUDA Driver Version / Runtime Version 6.5 / 6.5

Foram utilizadas dez configurações de base de dados para os nossos testes, as quais são descritas abaixo:

Configuração 1 padrão: $p =$ `LORD`, $|p| = 4$ - texto: Bíblia King James com conteúdo replicado, $|t| = 8\,703\,687$

Configuração 2 padrão: $p =$ `LORD`, $|p| = 4$ - texto: Bíblia King James, $|t| = 4\,351\,843$

Configuração 3 padrão: $p =$ `LORD`, $|p| = 4$ - texto: novo testamento da Bíblia King James, $|t| = 1\,017\,275$

Configuração 4 padrão: $p =$ `LORD`, $|p| = 4$ - texto: livro de Salmos da Bíblia King James, $|t| = 240\,885$

Configuração 5 padrão: $p =$ `LORD`, $|p| = 4$ - texto: livro de Gênesis Bíblia King James, $|t| = 166\,659$

Configuração 6 padrão: $p =$ `LORD`, $|p| = 4$ - texto: Primeira Epístola de João da Bíblia King James, $|t| = 13\,369$

Configuração 7 padrão: $p =$ `for his mercy endureth for ever`, $|p| = 31$ - texto: Bíblia King James com conteúdo replicado, $|t| = 8\,703\,687$

Configuração 8 padrão: $p =$ `for his mercy endureth for ever`, $|p| = 31$ - texto: Bíblia King James, $|t| = 4\,351\,843$

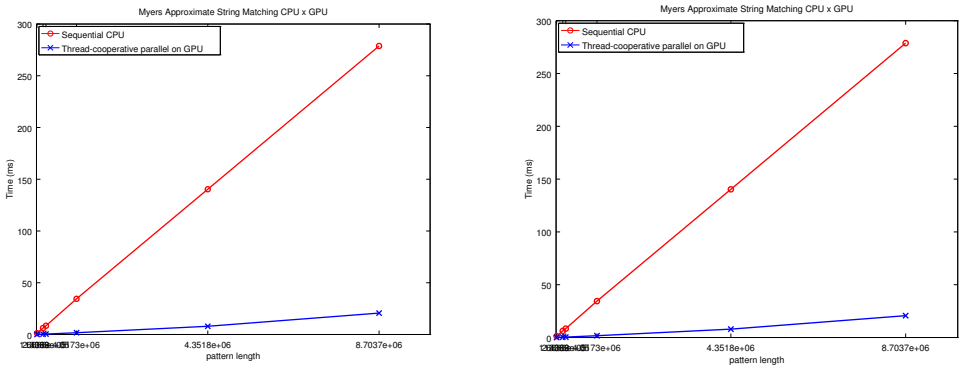
Configuração 9 padrão: $p =$ `for his mercy endureth for ever`, $|p| = 31$ - texto: novo testamento da Bíblia King James, $|t| = 1\,017\,275$

Configuração 10 padrão: $p =$ `for his mercy endureth for ever`, $|p| = 31$ - texto: livro de Salmos Bíblia King James, $|t| = 240\,885$

Configuração 11 padrão: $p =$ `for his mercy endureth for ever`, $|p| = 31$ - texto: livro de Gênesis Bíblia King James, $|t| = 166\,659$

Configuração 12 padrão: $p =$ `for his mercy endureth for ever`, $|p| = 31$ - texto: Primeira Epístola de João da Bíblia King James, $|t| = 13\,369$

Neste trabalho, estamos considerando o tempo do algoritmo de Myers executado em sequencial na CPU como sendo o tempo de parede, não sendo levado em conta o tempo de leitura do arquivo do padrão e do texto, e nem o tempo de impressão dos resultados. Já o tempo de execução da nossa abordagem paralela em GPU consiste do somatório de todos os tempos do *Profiling results* retornado pelo *nvprof*, não sendo considerados os tempos de leitura do padrão e do texto, como também não sendo considerados os tempos de chamadas de API do CUDA.



(a) $|p| = 4$ (b) $|p| = 31$

Figura 1: Comparação do tempo de execução sequencial em CPU com o tempo de execução da nossa abordagem *Thread-cooperative* em GPU.

4.2 Análise dos resultados

A Tabela 4.1 e a Figura 1 apresentam os resultados obtidos em nossa estratégia de paralelismo em GPU. Como pode ser percebido obtemos um considerável ganho de desempenho se comparado ao tempo de execução em CPU, de até 3 375% de melhora. Padrões maiores fazem com que menos *threads* sejam lançadas, o que prejudica o desempenho, conforme podemos verificar nos experimentos.

Nesta seção faremos uma breve análise dos resultados descritos através do uso do *nvprof*.

Configuração	Sequencial CPU (ms)	*Thread-cooperative* em GPU (ms)	*Speed-up*
Configuração 1	278,787018	20,677804	13,48
Configuração 2	140,347000	7,93738	17,68
Configuração 3	34,472000	1,705135	20,21
Configuração 4	8,427000	0,414089	20,35
Configuração 5	6,247000	0,298561	20,92
Configuração 6	1,269000	0,0376	33,75
Configuração 7	279,370026	15,87318	17,60
Configuração 8	139,198013	10,337652	13,46
Configuração 9	35,797001	2,380277	15,03
Configuração 10	8,577001	0,546658	15,68
Configuração 11	6,340000	0,390404	16,23
Configuração 12	1,460000	0,101409	14,39

Tabela 4.1: Análise do *Speed-up* da nossa abordagem *Thread-cooperative* em GPU com relação ao tempo de execução sequencial em CPU.

Resultados obtidos da execução da Configuração 2:

```
 nvprof ./myers -P Texts/pattern.txt -T Texts/bible.txt
==6608== NVPROF is profiling process 6608, command: ./myers -P Texts/pattern.txt -T Texts/bible.txt

Slices: 1048576 - Threads: 1024 - Blocks: 1024
Lenght of text: 4351843 - Slice base: 4 - Remaining: 157539
Kernel time 2.632000 (ms)
Total time 254.050018 (ms)
==6608== Profiling application: ./myers -P Texts/pattern.txt -T Texts/bible.txt
==6608== Profiling result:
Time(%)      Time     Calls       Avg       Min       Max  Name
 36.47%  2.8948ms         1  2.8948ms  2.8948ms  2.8948ms  [CUDA memcpy HtoD]
 34.78%  2.7605ms         1  2.7605ms  2.7605ms  2.7605ms  [CUDA memcpy DtoH]
 24.45%  1.9408ms         1  1.9408ms  1.9408ms  1.9408ms  myers_kernel(unsigned int, unsigned int, unsigned int,
     unsigned int, unsigned int, unsigned int, unsigned char*, char const *, unsigned int, unsigned int, unsigned
     int)
  4.25%  337.12us         1  337.12us  337.12us  337.12us  [CUDA memset]
  0.05%  4.1600us         1  4.1600us  4.1600us  4.1600us  [CUDA memcpy HtoA]

==6608== API calls:
Time(%)      Time     Calls       Avg       Min       Max  Name
 92.16%  128.52ms         1  128.52ms  128.52ms  128.52ms  cudaMallocArray
  4.05%  5.6431ms         2  2.8215ms  2.6440ms  2.9991ms  cudaMemcpy
  1.81%  2.5202ms         1  2.5202ms  2.5202ms  2.5202ms  cudaDeviceSynchronize
  0.87%  1.2068ms         1  1.2068ms  1.2068ms  1.2068ms  cudaDeviceReset
  0.48%  669.51us        83  8.0660us  1.9150us  232.29us  cuDeviceGetAttribute
  0.20%  273.37us         2  136.69us  133.73us  139.64us  cudaMalloc
  0.11%  157.76us         2  78.879us  74.863us  82.896us  cudaFree
  0.09%  129.02us         1  129.02us  129.02us  129.02us  cudaFreeArray
  0.06%  78.511us         1  78.511us  78.511us  78.511us  cuDeviceTotalMem
  0.05%  73.577us         1  73.577us  73.577us  73.577us  cudaLaunch
  0.04%  61.197us         1  61.197us  61.197us  61.197us  cuDeviceGetName
  0.02%  26.874us         1  26.874us  26.874us  26.874us  cudaMemset
  0.02%  23.799us         1  23.799us  23.799us  23.799us  cudaMemcpyToArray
  0.02%  21.680us         1  21.680us  21.680us  21.680us  cudaSetDevice
  0.01%  12.189us        11  1.1080us     687ns  3.4890us  cudaSetupArgument
  0.01%  9.8520us         1  9.8520us  9.8520us  9.8520us  cudaBindTextureToArray
  0.00%  6.3860us         2  3.1930us  2.2110us  4.1750us  cuDeviceGetCount
  0.00%  4.4400us         2  2.2200us  2.1350us  2.3050us  cuDeviceGet
```

```
 0.00%  4.0210us         1  4.0210us  4.0210us  4.0210us  cudaGetChannelDesc
 0.00%  2.7160us         1  2.7160us  2.7160us  2.7160us  cudaCreateChannelDesc
 0.00%  2.0740us         1  2.0740us  2.0740us  2.0740us  cudaConfigureCall

 nvprof --events warps_launched,global_ld_mem_divergence_replays,global_st_mem_divergence_replays ./myers -P Texts/
     pattern.txt -T Texts/bible.txt
==6657== NVPROF is profiling process 6657, command: ./myers -P Texts/pattern.txt -T Texts/bible.txt

Slices: 1048576 - Threads: 1024 - Blocks: 1024
Lenght of text: 4351843 - Slice base: 4 - Remaining: 157539
Kernel time 5.064000 (ms)
Total time 240.728012 (ms)
==6657== Profiling application: ./myers -P Texts/pattern.txt -T Texts/bible.txt
==6657== Profiling result:
==6657== Event result:
Invocations                                Event Name         Min         Max         Avg
Device "GeForce GT 740M (0)"
        Kernel: myers_kernel(unsigned int, unsigned int, unsigned int, unsigned int, unsigned int, unsigned int,
            unsigned char*, char const *, unsigned int, unsigned int, unsigned int)
          1                            warps_launched       32768       32768       32768
          1          global_ld_mem_divergence_replays      130302      130302      130302
          1          global_st_mem_divergence_replays          13          13          13

nvprof --metrics sm_efficiency,warp_execution_efficiency,tex_cache_hit_rate,l1_cache_global_hit_rate,
    l2_texture_read_hit_rate,l2_l1_read_hit_rate,l2_utilization,tex_utilization ./myers -P Texts/pattern.txt -T
    Texts/bible.txt
==6771== NVPROF is profiling process 6771, command: ./myers -P Texts/pattern.txt -T Texts/bible.txt

Slices: 1048576 - Threads: 1024 - Blocks: 1024
==6771== Warning: Some kernel(s) will be replayed on device 0 in order to collect all events/metrics.
Lenght of text: 4351843 - Slice base: 4 - Remaining: 157539
Kernel time 105.356003 (ms)
Total time 335.818024 (ms)
==6771== Profiling application: ./myers -P Texts/pattern.txt -T Texts/bible.txt
==6771== Profiling result:
==6771== Metric result:
Invocations                               Metric Name                         Metric Description         Min
    Max         Avg
Device "GeForce GT 740M (0)"
        Kernel: myers_kernel(unsigned int, unsigned int, unsigned int, unsigned int, unsigned int, unsigned int,
            unsigned char*, char const *, unsigned int, unsigned int, unsigned int)
```

```
1                    l1_cache_global_hit_rate                    L1 Global Hit Rate       0.00%
    0.00%       0.00%
1                              sm_efficiency                Multiprocessor Activity      99.10%
    99.10%      99.10%
1                       tex_cache_hit_rate                Texture Cache Hit Rate      97.55%
    97.55%      97.55%
1                      l2_l1_read_hit_rate                L2 Hit Rate (L1 Reads)      87.52%
    87.52%      87.52%
1                    l2_texture_read_hit_rate          L2 Hit Rate (Texture Reads)      83.87%
    83.87%      83.87%
1                      warp_execution_efficiency          Warp Execution Efficiency      100.00%
    100.00%     100.00%
1                              l2_utilization                L2 Cache Utilization     Mid (4)     Mid
    (4)     Mid (4)
1                             tex_utilization               Texture Cache Utilization     Mid (6)     Mid
    (6)     Mid (6)
```

Resultados obtidos da execução da Configuração 8:


```
 nvprof ./myers -P Texts/pattern2.txt -T Texts/bible.txt
==3460== NVPROF is profiling process 3460. command: ./myers -P Texts/pattern2.txt -T Texts/bible.txt

Slices: 131072 - Threads: 1024 - Blocks: 128
Lenght of text: 4351843 - Slice base: 33 - Remaining: 26467
Kernel time 5.063000 (ms)
Total time 142.764999 (ms)
==3460== Profiling application: ./myers -P Texts/pattern2.txt -T Texts/bible.txt
==3460== Profiling result:
Time(%)      Time     Calls       Avg       Min       Max  Name
 42.27%  4.3701ms         1  4.3701ms  4.3701ms  4.3701ms  myers_kernel(unsigned int. unsigned int. unsigned int.
    unsigned int. unsigned int. unsigned int. unsigned char*. char const *. unsigned int. unsigned int. unsigned
    int)
 27.88%  2.8823ms         1  2.8823ms  2.8823ms  2.8823ms  [CUDA memcpy HtoD]
 26.55%  2.7449ms         1  2.7449ms  2.7449ms  2.7449ms  [CUDA memcpy DtoH]
  3.25%  336.16us         1  336.16us  336.16us  336.16us  [CUDA memset]
  0.04%  4.1920us         1  4.1920us  4.1920us  4.1920us  [CUDA memcpy HtoA]

==3460== API calls:
Time(%)      Time     Calls       Avg       Min       Max  Name
```

```
 90.81%  121.74ms         1  121.74ms  121.74ms  121.74ms  cudaMallocArray
  4.36%  5.8387ms         2  2.9193ms  2.5828ms  3.2559ms  cudaMemcpy
  3.69%  4.9440ms         1  4.9440ms  4.9440ms  4.9440ms  cudaDeviceSynchronize
  0.29%  384.34us         1  384.34us  384.34us  384.34us  cudaDeviceReset
  0.25%  339.66us        83  4.0920us     596ns  131.96us  cuDeviceGetAttribute
  0.20%  271.02us         2  135.51us  130.53us  140.49us  cudaMalloc
  0.11%  153.78us         2  76.892us  72.055us  81.729us  cudaFree
  0.10%  129.73us         1  129.73us  129.73us  129.73us  cudaFreeArray
  0.06%  78.394us         1  78.394us  78.394us  78.394us  cudaLaunch
  0.03%  43.799us         1  43.799us  43.799us  43.799us  cuDeviceGetName
  0.03%  43.508us         1  43.508us  43.508us  43.508us  cuDeviceTotalMem
  0.02%  26.418us         1  26.418us  26.418us  26.418us  cudaMemset
  0.02%  23.217us         1  23.217us  23.217us  23.217us  cudaMemcpyToArray
  0.01%  12.865us        11  1.1690us     772ns  3.7590us  cudaSetupArgument
  0.01%  9.6100us         1  9.6100us  9.6100us  9.6100us  cudaBindTextureToArray
  0.01%  7.6510us         1  7.6510us  7.6510us  7.6510us  cudaSetDevice
  0.00%  4.1310us         1  4.1310us  4.1310us  4.1310us  cudaGetChannelDesc
  0.00%  2.6830us         1  2.6830us  2.6830us  2.6830us  cudaConfigureCall
  0.00%  2.3210us         2  1.1600us     746ns  1.5750us  cuDeviceGetCount
  0.00%  1.5690us         2     784ns     753ns     816ns  cuDeviceGet
  0.00%     868ns         1     868ns     868ns     868ns  cudaCreateChannelDesc

 nvprof --events warps_launched.global_ld_mem_divergence_replays.global_st_mem_divergence_replays ./myers -P Texts/
    pattern2.txt -T Texts/bible.txt
==3508== NVPROF is profiling process 3508. command: ./myers -P Texts/pattern2.txt -T Texts/bible.txt

Slices: 131072 - Threads: 1024 - Blocks: 128
Lenght of text: 4351843 - Slice base: 33 - Remaining: 26467
Kernel time 7.520000 (ms)
Total time 246.909012 (ms)
==3508== Profiling application: ./myers -P Texts/pattern2.txt -T Texts/bible.txt
==3508== Profiling result:
==3508== Event result:
Invocations                                Event Name         Min         Max         Avg
Device "GeForce GT 740M (0)"
        Kernel: myers_kernel(unsigned int. unsigned int. unsigned int. unsigned int. unsigned int. unsigned int.
            unsigned char*. char const *. unsigned int. unsigned int. unsigned int)
          1                            warps_launched        4096        4096        4096
          1          global_ld_mem_divergence_replays     2081757     2081757     2081757
          1          global_st_mem_divergence_replays           0           0           0
```

```
 nvprof --metrics sm_efficiency,warp_execution_efficiency,tex_cache_hit_rate,l1_cache_global_hit_rate,
     l2_texture_read_hit_rate,l2_l1_read_hit_rate,l2_utilization,tex_utilization ./myers -P Texts/pattern2.txt -T
     Texts/bible.txt
==3576== NVPROF is profiling process 3576, command: ./myers -P Texts/pattern2.txt -T Texts/bible.txt

Slices: 131072 - Threads: 1024 - Blocks: 128
==3576== Warning: Some kernel(s) will be replayed on device 0 in order to collect all events/metrics.
Lenght of text: 4351843 - Slice base: 33 - Remaining: 26467
Kernel time 171.020004 (ms)
Total time 414.281006 (ms)
==3576== Profiling application: ./myers -P Texts/pattern2.txt -T Texts/bible.txt
==3576== Profiling result:
==3576== Metric result:
Invocations                               Metric Name                        Metric Description         Min
    Max         Avg
Device "GeForce GT 740M (0)"
        Kernel: myers_kernel(unsigned int, unsigned int, unsigned int, unsigned int, unsigned int, unsigned int,
            unsigned char*, char const *, unsigned int, unsigned int, unsigned int)
          1                 l1_cache_global_hit_rate                  L1 Global Hit Rate       0.00%
            0.00%       0.00%
          1                            sm_efficiency              Multiprocessor Activity      99.91%
            99.91%      99.91%
          1                      tex_cache_hit_rate                Texture Cache Hit Rate      92.61%
            92.61%      92.61%
          1                     l2_l1_read_hit_rate                L2 Hit Rate (L1 Reads)      98.36%
            98.36%      98.36%
          1                l2_texture_read_hit_rate          L2 Hit Rate (Texture Reads)     100.00%
            100.00%     100.00%
          1                  warp_execution_efficiency           Warp Execution Efficiency     100.00%
            100.00%     100.00%
          1                           l2_utilization                L2 Cache Utilization    Max (10)    Max
            (10)    Max (10)
          1                          tex_utilization              Texture Cache Utilization     Low (3)     Low
            (3)     Low (3)
```

A partir dos resultados do *nvprof*, observe que um dos nosso maiores gargalos é o acesso divergente à memória global, tanto para leitura quanto para escrita. Empiricamente, nos nossos testes pudemos perceber que este acesso divergente é exclusivamente devido ao armazenamento das ocorrências do padrão no texto. Para tanto, utilizamos um vetor armazenado na memória global onde casa *thread* armazena na posição i do vetor a ocorrência ou não do padrão no texto na posição i (1 para ocorreu, 0 para não ocorreu), prevalecendo o resultado de ocorrência (isto é, o valor 1 predomina com relação ao valor 0). Claramente isso pode ocasionar divergência de *branch*, no entanto este é mínimo, pois através dos resultados do *nvprof* podemos perceber que a eficiência de *warp* é de 100%. No entanto, esta estratégia para armazenar os valores de ocorrência gera um alto acesso não divergente à memória global, o que afeta, e muito, no desempenho da nossa aplicação (é deixado como trabalhos futuros otimizar este acesso).

Uma outra observação válida a se fazer é com relação à perca de desempenho ao aumentarmos o tamanho do padrão. Isso é ocasionado, pois a quantidade de fatias em que o texto será dividido é menor (veja (3.2)), e consequentemente o número de *threads* lançados também será menor. No entanto, a carga de trabalho total continua a mesma (já que o tamanho do texto não foi alterado), aumentando portanto o trabalho a ser realizado por cada *thread*. Com isso, o tempo de execução de cada *thread* é maior, o que ocasiona a perca de desempenho da aplicação como um todo.

5 Otimizações realizadas

Com o intutito de melhorar o desempenho da nossa abordagem *Thread-cooperative* em GPU para o problema de *string matching*, foram tomadas algumas medidas comuns de otimização em nossa implementação, as quais descrevemos nesta seção.

Primeiramente, sempre que possível, foi realizado o uso da memória constante da GPU, com o intuito de se evitar ao máximo o acesso à memória global, que de modo geral possui latência muito maior do que o da memória constante.

Além disso, armazenamos a matriz PEq utilizando memória de textura (já que esta é utilizada na GPU somente para leitura), com o intuito de reduzir o alto acesso à memória global, e consequentemente diminuir também a latência de acesso aos dados, uma vez que, de modo geral, o acesso à memória de textura é consideravelmente mais rápido do que o acesso à memória global.

Por fim, vale destacar também que foi evitado ao máximo usar estruturas de desvio que pudessem ocasionar divergência de *branch*, tendo em vista que

a ocorrência desses serializa as *threads* respectivas à execução, o que leva a um alto impacto (negativamente) no desempenho da nossa aplicação.

6 Conclusão e Trabalhos Futuros

Como podemos perceber através dos resultados descritos na Seção 4.2, a nossa abordagem de paralelismo em GPU apresentou uma melhora significativa do tempo de execução se comparado ao tempo sequencial em CPU. Desta forma, podemos considerar que é válido se investir nesta abordagem buscando realizar tantas outras otimizações na implementação do código. Dentre tais otimizações já podemos citar algumas: armazenar o vetor referente ao texto utilizando memória compartilhada (espera-se que a aplicação obtenha um ganho de desempenho consideravelmente alto ao se realizar esta mudança) e buscar uma alternativa melhor de armazenar os resultados de ocorrência do padrão no texto (conforme descrevemos na Seção 4.2, um dos grandes gargalos da nossa aplicação é o armazenamento desses valores, então é esperado que uma melhoria nesta forma de escrita irá gerar um alto ganho de desempenho).

Referências Bibliográficas

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

[2] Hyyrö, H. Explaining and extending the bit-parallel approximate string matching algorithm of myers. Relatório técnico, Departament of Computer And Information Sciences, University of Tamper, Finland, 2001.

[3] Hyyrö, H. Bit-parallel approximate string matching algorithms with transposition. In *In Proc. 13th Combinatorial Pattern Matching (CPM 2002), LNCS 2373* (2002), pp. 203–224.

[4] Levenshtein, V. Binary codes capable of correcting deletions, insertions, and reversals. *Soviet Physics Doklady 10*, 8 (1966), 707–710.

[5] Myers, G. A fast bit-vector algorithm for approximate string matching based on dynamic programming. *J. ACM 46*, 3 (maio de 1999), 395–415.

[6] Owens, J. D., Luebke, D., Govindaraju, N., Harris, M., Krüger, J., Lefohn, A. E., e Purcell, T. A survey of general-purpose computation on graphics hardware, 2007.

[7] Ukkonen, E. Finding approximate patterns in strings. *Journal of algorithms 6*, 1 (1985), 132–137.