Universidade de Brasília
Instituto de Ciências Exatas
Departamento de Ciência da Computação

Fickett-CUDAlign: Comparação Paralela de Sequências Biológicas com Estratégia Multi-Bloco de Faixas Ajustáveis

Gabriel Heleno Gonçalves da Silva

Dissertação apresentada como requisito parcial
para conclusão do Mestrado em Informática

Orientadora
Prof.ª Dr.ª Alba Cristina Magalhães Alves de Melo

Brasília
2016

Universidade de Brasília — UnB
Instituto de Ciências Exatas
Departamento de Ciência da Computação
Programa de pós-graduação em Informática

Coordenadora: Prof.ª Dr.ª Célia Ghedini Ralha

Banca examinadora composta por:

Prof.ª Dr.ª Alba Cristina Magalhães Alves de Melo (Orientadora) — CIC/UnB
Prof. Dr. Fábio Moreira Costa — INF/UFG
Prof. Dr. George Luiz Medeiros Teodoro — CIC/UnB

CIP — Catalogação Internacional na Publicação

Gabriel Heleno Gonçalves da Silva, .

Fickett-CUDAlign: Comparação Paralela de Sequências Biológicas com Estratégia Multi-Bloco de Faixas Ajustáveis / Gabriel Heleno Gonçalves da Silva. Brasília : UnB, 2016.
83 p. : il. ; 29,5 cm.

Dissertação (Mestrado) — Universidade de Brasília, Brasília, 2016.

1. Processamento Paralelo, 2. Alinhamento Ótimo de Sequências,
3. Algoritmo de Fickett

CDU 004.4

Endereço: Universidade de Brasília
Campus Universitário Darcy Ribeiro — Asa Norte
CEP 70910-900
Brasília–DF — Brasil

Universidade de Brasília
Instituto de Ciências Exatas
Departamento de Ciência da Computação

Fickett-CUDAlign: Comparação Paralela de Sequências Biológicas com Estratégia Multi-Bloco de Faixas Ajustáveis

Gabriel Heleno Gonçalves da Silva

Dissertação apresentada como requisito parcial
para conclusão do Mestrado em Informática

Prof.ª Dr.ª Alba Cristina Magalhães Alves de Melo (Orientadora)
CIC/UnB

Prof. Dr. Fábio Moreira Costa
INF/UFG

Prof. Dr. George Luiz Medeiros Teodoro
CIC/UnB

Prof.ª Dr.ª Célia Ghedini Ralha
Coordenadora do Programa de pós-graduação em Informática

Brasília, 22 de Março de 2016

Agradecimentos

Agradeço primeiramente a Deus, por me dar a oportunidade de fazer o mestrado e guiar meus passos durante todo esse percurso até aqui. Eu sei que a cada dia é a Sua graça que me dá forças para alcançar a realização dos meus sonhos.

À minha família, agradeço pelo apoio incondicional e pela compreensão e auxílio em todas as minhas dificuldades. Vocês são e sempre serão um grande exemplo para mim, sem vocês nada do que foi feito faria sentido.

Agradeço à minha orientadora pela paciência e dedicação. Durante todas as nossas reuniões pude aprender um pouco mais do que é realmente ser um mestre. Os seus ensinamentos ampliaram os meus horizontes e se consolidaram nessa dissertação. Aos professores do departamento, obrigado por compartilhar o seu conhecimento e experiência, tanto acadêmica como de vida.

Agradeço aos meus amigos, que sempre acreditaram em mim. As lembranças de tudo o que vivemos juntos me dão forças para continuar independentemente da situação. Obrigado por fazerem parte da minha vida.

Enfim, minha sincera gratidão a todos que de alguma forma me ajudaram a superar os meus limites, pois todos os seus esforços me fizeram crescer e me tornar uma pessoa melhor.

Resumo

A comparação de sequências biológicas é uma operação importante na Bioinformática, que é realizada frequentemente. Os algoritmos exatos para comparação de sequências obtêm o resultado ótimo calculando uma ou mais matrizes de programação dinâmica. Estes algoritmos têm complexidade de tempo $O(mn)$, onde m e n são os tamanhos das sequências. Fickett propôs um algoritmo que é capaz de reduzir a complexidade para $O(kn)$, onde k é a faixa de computação e representa a quantidade de diagonais da matriz efetivamente calculadas. Nessa dissertação de mestrado, propomos e avaliamos o Fickett-CUDAlign, uma estratégia paralela que divide a comparação de sequências em múltiplas comparações de subsequências e calcula uma faixa de Fickett apropriada para cada comparação de sequência (bloco). Com esta abordagem, nós reduzimos potencialmente o número de células calculadas, quando comparada ao Fickett, que usa uma única faixa para toda a comparação. Nossa estratégia multi-bloco ajustável foi programada em C/C++ e *pthreads* e foi integrada ao estágio 4 do CUDAlign, uma ferramenta do estado da arte para comparações ótimas de sequências biológicas. O Fickett-CUDAlign foi usado para comparar sequências reais de DNA cujo tamanho variou de 10KBP (Milhares de Pares de Base) a 47MBP (Milhões de Pares de Base), alcançando um *speedup* de 59,60x na comparação 10MBP x 10MBP, quando comparado ao estágio 4 do CUDAlign. Neste caso, o tempo de execução foi reduzido de 53,56 segundos para 0,90 segundo.

Palavras-chave: Processamento Paralelo, Alinhamento Ótimo de Sequências, Algoritmo de Fickett

Abstract

Biological sequence comparison is an important task in Bioinformatics, which is frequently performed. The exact algorithms for sequence comparison obtain the optimal result by calculating one or more dynamic programming matrices. These algorithms have $O(mn)$ time complexity, where m and n are the sizes of the sequences. Fickett proposed an algorithm which is able to reduce time complexity to $O(kn)$, where k is the computation band and represents the amount of matrix diagonals actually calculated. In this MSc Dissertation, we propose and evaluate Fickett-CUDAlign, a parallel strategy that splits a pairwise sequence comparison in multiple comparisons of subsequences and calculates an appropriate Fickett band to each subsequence comparison (block). With this approach, we potentially reduce the number of cells calculated, when compared to Fickett, which uses a unique band to the whole comparison. Our adjustable multi-block strategy was programmed in C/C++ and pthreads and was integrated to the stage 4 of CUDAlign, a state-of-the-art tool for optimal biological sequence comparison. Fickett-CUDAlign was used to compare real DNA sequences whose sizes ranged from 10KBP (Thousands of Base Pairs) to 47MBP (Millions of Base Pairs), reaching a speedup of 59.60x in the 10MBP x 10MBP comparison, when compared to CUDAlign's stage 4. In this case, the execution time was reduced from 53.56 seconds to 0.90 second.

Keywords: Parallel Processing, Optimal Sequence Alignment, Fickett's Algorithm

Sumário

Lista de Figuras

Lista de Tabelas

Capítulo 1

Introdução

Atualmente, observa-se um aumento exponencial no número de sequências biológicas armazenadas em bancos de dados genômicos. Esse "dilúvio de dados" [31] foi ocasionado por uma impressionante evolução nas técnicas de sequenciamento de organismos, que deu origem à tecnologia conhecida como NGS (*Next Generation Sequencing*), aumentando substancialmente o número de sequências produzidas no mesmo tempo em relação à sua predecessora [21].

Quando um organismo novo tem o seu código genético sequenciado, é importante que suas características funcionais sejam descobertas. Geralmente isso é feito a partir da comparação do seu código genético com o de outros organismos, em busca de similaridades. A comparação de sequências biológicas é então uma das operações básicas da Bioinformática. Em cada comparação, um escore indica o grau de semelhança entre as sequências [21]. Além do escore, também é geralmente produzido o alinhamento, no qual uma sequência é colocada sobre a outra de modo a ressaltar as similaridades/diferenças [13]. No alinhamento, espaços (*gaps*) podem ser inseridos para produzir melhores resultados.

O algoritmo exato mais utilizado para comparar duas sequências é o proposto por Smith-Waterman (SW) [33], baseado em técnicas de programação dinâmica, com complexidade quadrática de tempo e espaço. O algoritmo SW utiliza uma matriz de programação dinâmica de tamanho $m + 1$ x $n + 1$ para comparar duas sequências de tamanho m e n. Embora ele obtenha o resultado ótimo, necessita de muito poder de computação para ser executado em um tempo razoável. Para diminuir essa exigência de poder de processamento, foram criadas soluções heurísticas que, com o prejuízo de o resultado não ser ótimo, diminuem consideravelmente o tempo de execução [21].

A computação de alto desempenho (*High Performance Computing - HPC*) pode ser utilizada para acelerar a execução dos algoritmos exatos de comparação de sequências. De fato, foram propostas nas últimas décadas várias técnicas para executar o algoritmo

SW em *clusters*[4] e *grids* [15]. Especificamente, aceleradores como as GPUs (*Graphics Processing Unit*) e FPGAs (*Field Programmable Gate Arrays*) vêm sendo utilizados na execução do algoritmo SW [16][19]. Além disso, instruções vetoriais dos processadores que compõem os multicores, como as instruções SSE da Intel®, também têm sido utilizadas para acelerar a execução do SW [8][27].

O CUDAlign [6] é uma ferramenta em GPU para a comparação de sequências longas de DNA com o algoritmo SW. Ele é executado em 5 etapas, onde a etapa 1 executa a fase 1 do SW (obter o escore ótimo) e as etapas 2 a 5 executam a fase 2 do SW (obter o alinhamento ótimo).

A fase 1 do algoritmo SW já foi bastante otimizada [28][6][8][16], pois verificou-se que essa era a fase mais demorada. No entanto, ao se executar a etapa 1 do CUDAlign com 64 GPUs [5], verificou-se que o tempo de execução dessa etapa decresceu de maneira próxima do linear, enquanto o tempo de execução das etapas 2 a 5 do CUDAlign permaneceu constante, tornando-se um limitante no desempenho.

Em [30] foi proposta uma otimização para a etapa 2 do CUDAlign, que conseguiu reduzir em até 21x o tempo dessa etapa e também conseguiu realizar a execução das etapas 2, 3 e 4 em *pipeline*. Isso permitiu reduzir o tempo de *traceback* de 3508 segundos para 167 segundos. Mesmo assim, o tempo da etapa 4 continuou alto na comparação de outras sequências.

Uma das maneiras para se reduzir o tempo de execução do SW é aplicar otimizações estruturais na matriz de programação dinâmica, dentre as quais a utilização do método de Fickett [9], que permite estabelecer uma faixa de valores que serão computados na matriz de programação dinâmica. Foi provado que o método de Fickett reduz a complexidade de tempo e memória para $O(kn)$, onde n é o tamanho das sequências e k o tamanho da faixa [9]. Essa redução de tempo de execução é obtida através da redução da área de computação da matriz de programação dinâmica para uma região que compreenda o alinhamento.

Para a redução da memória necessária ao algoritmo SW, utiliza-se comumente o algoritmo proposto por Hirschberg [14], que obtém de maneira recursiva os pontos que fazem parte do alinhamento ótimo, partindo de um ponto inicial na coluna do meio da matriz de programação dinâmica. O algoritmo de Hirschberg foi adaptado por Myers-Miller [22] para computar um modelo mais complexo de colunas, chamado *affine gap*.

A presente Dissertação de Mestrado tem por objetivo propor e avaliar uma estratégia que permita a execução multi-bloco do algoritmo de Fickett em conjunto com o algoritmo de Myers-Miller, com faixas de tamanho ajustável. Com isso, o tamanho da faixa é ajustado de acordo com o comportamento do alinhamento em sub-regiões, reduzindo a área computada em relação ao algoritmo de Fickett, que utiliza uma faixa única em

toda a matriz de programação dinâmica. A estratégia proposta foi implementada no CUDAlign [6], substituindo sua fase 4, que executa o algoritmo de Myers-Miller.

Resultados obtidos com sequências reais de DNA de tamanho variando de 10KBP (Milhares de Pares de Bases) a 47MBP (Milhões de Pares de Bases) mostram que a estratégia proposta é capaz de acelerar a fase 4 em até 59,60 vezes, quando comparado com a fase 4 atual do CUDAlign. Nesse caso o tempo de execução foi reduzido de 53,56 segundos para 0,90 segundo.

Esta dissertação está dividida nos seguintes capítulos. No Capítulo 2, o alinhamento de sequências biológicas é abordado e os algoritmos Needleman-Wunsch, Smith-Waterman, Myers-Miller e Fickett são detalhados. No Capítulo 3, algumas ferramentas que realizam a paralelização de sequências biológicas em plataformas de alto desempenho são analisadas e comparadas. No Capítulo 4, é apresentada a estratégia multi-bloco com faixas de tamanho variável e no Capítulo 5 os resultados obtidos são discutidos. Finalmente, o Capítulo 6 apresenta a conclusão e sugere trabalhos futuros.

Capítulo 2

Alinhamento de Sequências Biológicas

Uma sequência biológica é uma sequência de caracteres que constituem RNA, DNA ou proteínas [21]. As sequências de RNA e DNA são compostas por nucleotídeos, enquanto as proteínas são compostas por aminoácidos. O alfabeto dos nucleotídeos que compõem as moléculas de DNA e RNA são: A, C, G, T, e A, C, G, U, respectivamente. Já o alfabeto dos aminoácidos é formado basicamente por 20 elementos, que são: A, R, N, D, E, C, G, Q, H, I, L, K, M, F, P, S, Y, T, W, V.

A finalidade da comparação de sequências biológicas é evidenciar similaridades entre os organismos de modo a permitir que funcionalidades, estruturas e aspectos evolutivos sejam descobertos [21]. Isso se deve ao fato de que muitas sequências similares possuem a mesma função biológica e, ao comparar sequências com funcionalidades conhecidas com sequências ainda não estudadas, é possível identificar regiões similares, indicando uma característica comum, ou regiões que se modificaram de uma sequência para outra.

O objetivo da comparação de sequências biológicas é obter um escore, que mede a similaridade entre as sequências e um alinhamento, que evidencia as regiões de similaridade/diferenças entre as sequências [13]. Nas Seções 2.1 a 2.4 são detalhados os algoritmos exatos mais utilizados para a comparação de sequências.

2.1 Algoritmo Needleman-Wunsch (NW)

Um alinhamento é dito global quando são utilizados todos os caracteres das sequências no alinhamento [32]. Para encontrar os alinhamentos globais, investigam-se os escores gerados pelas operações de tentativa de pareamento, sendo que os elementos podem coincidir ou não. Quando ocorre a coincidência (*match*), ela geralmente é contabilizada positivamente e quando a coincidência não ocorre (*mismatch*), é contabilizada negativamente. Também são penalizadas as inserções e remoções através da introdução de *gaps* [32].

Em [23], Needleman e Wunsch propuseram um algoritmo exato baseado em programação dinâmica para o problema do alinhamento global. O algoritmo é executado em duas fases: (1) cálculo da matriz de programação dinâmica e (2) *backtracking*.

Na fase 1, é usada programação dinâmica para alinhar os prefixos das sequências. Na medida em que prefixos são alinhados, escores parciais são utilizados para calcular os escores para os prefixos maiores. Assim, a cada passo, dois caracteres (um de cada sequência) são comparados. O resultado da comparação é utilizado conjuntamente com os escores já obtidos com o alinhamento prévio dos prefixos para se obter o valor atual.

O algoritmo NW considera três casos [23]:

a) Alinhar os dois caracteres, podendo haver coincidência ou não;

b) Alinhar um espaço da primeira sequência com um elemento da segunda sequência, correspondendo a uma remoção na primeira sequência ou uma inserção na segunda;

c) Alinhar um caractere da primeira sequência com um espaço da segunda sequência, correspondendo a uma inserção na primeira sequência ou uma remoção na segunda;

Sejam duas sequências $A = a_1, a_2, a_3, ...a_m$ e $B = b_1, b_2, b_3, ...b_n$, onde m e n são os tamanhos das respectivas sequências. Os escores parciais (correspondentes aos alinhamentos dos prefixos) são mantidos em uma matriz de programação dinâmica M, composta de m x n elementos $M[0..m, 0..n]$. Cada um dos elementos $M[i, j]$ da matriz é calculado utilizando as três possibilidades de alinhamento descritas em (a), (b) e (c). Estas operações são descritas pela equação de recorrência 2.1.

$$M(i,j) = max \begin{cases} M(i-1, j-1) + x(i,j) \\ M(i, j-1) + g \\ M(i-1, j) + g \end{cases} \tag{2.1}$$

onde g é a pontuação para *gaps* e $x(i, j)$ é a pontuação para *matches*/*mismatches*. Note que, de acordo com a equação de recorrência, o cálculo de cada célula $M[i, j]$ depende do valor de três células previamente calculadas: $M[i-1, j]$, $M[i-1, j-1]$, $M[i, j-1]$.

No caso de DNA ou RNA, geralmente é usada uma pontuação única para *match* e uma pontuação única para *mismatch*. No caso de proteínas, são usadas matrizes 20x20, chamadas matrizes de substituição, que contêm a pontuação de *match* e *mismatch* para cada par de aminoácidos [21].

Ao final da fase 1, a última célula à direita e abaixo ($M[m, n]$) conterá o escore ótimo do alinhamento global.

Na segunda fase (*backtracking*), é necessário descobrir a sequência de operações que levou ao escore ótimo. Para isto, é necessário saber qual das três células ($M[i-1, j], M[i-1, j-1], M[i, j-1]$) foi utilizada para o cálculo do escore naquela posição. Com esta informação, é possível percorrer o caminho reverso (*backtracking*) do elemento $M[m, n]$

até o elemento $M[0,0]$. Este caminho (ou caminhos, caso haja mais de um) corresponde ao(s) alinhamento(s) global(is) ótimo(s).

A complexidade de tempo do algoritmo Needleman-Wunsch é $O(mn)$ [23]. O espaço necessário depende de quais fases serão executadas. Para executar somente a fase 1, é necessário manter na memória apenas duas linhas (uma com o resultado anterior e outra que está sendo calculada). Assim, a complexidade de espaço é $O(n)$ onde n é o tamanho de uma sequência. No entanto, para recuperar também o alinhamento (fases 1 e 2), deve-se fazer o percurso reverso da matriz (*backtraking*) sobre os elementos da matriz. Isto nos leva à complexidade $O(mn)$ de espaço [21].

Na figura 2.1, é apresentado um exemplo de alinhamento global utilizando o algoritmo NW. As sequências a serem comparadas são A=TAGGAG e B=TGCTAG. Neste exemplo, as inserções de espaços (*gaps*) possuem escore -1, as coincidências (*matches*) recebem escore +1 e as substituições (*mismatches*) recebem escore -1. Os dois alinhamentos globais ótimos são apresentados na Figura 2.2. A seta na diagonal indica o alinhamento dos caracteres constantes na linha e coluna correspondentes. A seta vertical indica o alinhamento do caracter na vertical com um *gap*, enquanto a seta na horizontal indica o alinhamento do caracter na horizontal com um *gap*.

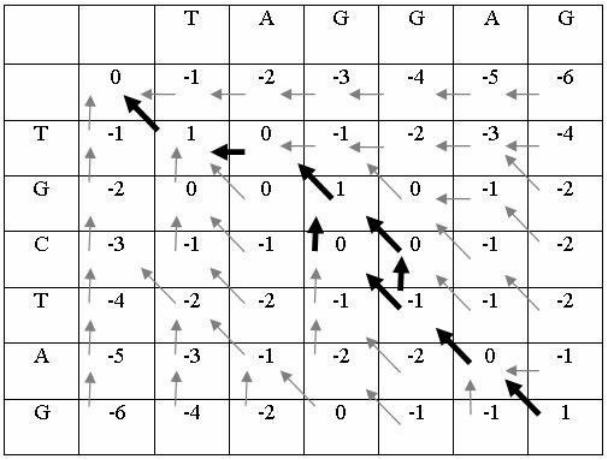


Figura 2.1: Matriz de programação dinâmica de Alinhamento Global com o algoritmo NW.

(1)
TAG-GAG
T -GCTAG

(2)
TAGG-AG
T- GCTAG

Figura 2.2: Alinhamentos globais ótimos produzidos na fase de *backtracking*.

2.2 Algoritmo Smith-Waterman (SW)

O alinhamento local difere do alinhamento global por alinhar trechos das sequências ao invés de alinhar as sequências inteiras, procurando por regiões de alta similaridade dentro das mesmas [32].

O algoritmo proposto por Smith-Waterman [33] é baseado no algoritmo de Needleman-Wunsch (Seção 2.1), porém adaptado para tratar o problema do alinhamento local. No algoritmo NW, o melhor alinhamento pode ter um escore bastante negativo, desde que seja o maior obtido. Para impedir isto, a equação 2.1 foi modificada impedindo valores menores que zero. Logo, como o Smith-Waterman tem caráter local, não são permitidos alinhamentos com um número muito grande de inserções, remoções e substituições. Isto é feito atribuindo um valor zero caso o escore se torne negativo. Isto faz com que o algoritmo busque a maior região de similaridade entre duas sequências.

O alinhamento local de Smith-Waterman [33], da mesma forma que NW, possui complexidade no tempo e no espaço de $O(nm)$. O cálculo de cada célula da matriz de programação dinâmica é feito, então, de acordo com a equação 2.2.

$$M(i,j) = max \begin{cases} 0 \\ M(i,j-1) + g \\ M(i-1,j-1) + x(i,j) \\ M(i-1,j) + g \end{cases} \tag{2.2}$$

onde $x(i,j)$ é a pontuação para *matches*/*mismatches* e g é a pontuação para *gaps*.

Além da introdução do valor zero na equação de recorrência e da inicialização da primeira linha e coluna da matriz M com valor zero, a recuperação do alinhamento local ótimo ocorre a partir da célula da matriz de programação dinâmica que possui o maior escore. A partir dessa célula, percorre-se o caminho inverso até encontrar uma célula que tenha valor zero. No alinhamento local, também é possível haver vários alinhamentos ótimos a partir de uma mesma célula.

A figura 2.3 apresenta um exemplo do alinhamento local utilizando o algoritmo de Smith-Waterman, utilizando também as sequências A=TAGGAG e B=TGCTAG, com as mesmas pontuações da Figura 2.1. O alinhamento local ótimo possui escore igual a 3 e é apresentado na Figura 2.4.

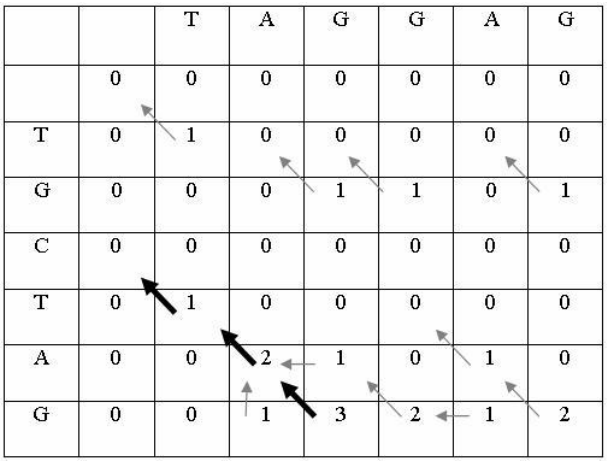


Figura 2.3: Matriz de programação dinâmica do Alinhamento Local com o algoritmo SW.

T A G
T A G

Figura 2.4: Alinhamento local ótimo produzido na fase de *backtracking*.

2.3 Algoritmo Gotoh

O algoritmo Gotoh [11] utiliza programação dinâmica para o alinhamento global de pares de sequências com o modelo *affine-gap* [21]. Nesse modelo, a penalidade para se iniciar uma sequência de *gaps* é maior do que a penalidade para estendê-la. O modelo *affine-gap* é interessante porque muitas vezes um número grande de *gaps* pode ser produzido por um único evento evolutivo.

Sejam $A = a_1, a_2, ..., a_m$ e $B = b_1, b_2, ..., b_n$ duas sequências. No modelo *affine-gap* um *gap* de tamanho k é da forma $w_k = uk + v$, com $u \leq 0$ e $v \leq 0$, onde v é a pontuação dada ao primeiro *gap* (G_open) e u é a pontuação dada aos *gaps* subsequentes (G_extend). O peso $x(i, j)$ é dado ao alinhamento do par s_i e t_j e, normalmente, $x(i, j) \geq 0$ se $s_i = t_j$ e $x(i, j) < 0$ se $s_i = t_j$.

Além da matriz M, necessita-se de duas outras matrizes (P e Q) para tratar sequências de *gaps* em cada uma das sequências. Sendo assim, as equações de recorrência de Gotoh são dadas pelas Equações 2.3, 2.4 e 2.5 [11].

$$M(i,j) = max \begin{cases} M(i-1, j-1) + x(i,j) \\ P(i,j) \\ Q(i,j) \end{cases} \tag{2.3}$$

$$P(i,j) = max \begin{cases} M(i-1,j) + w_1 \\ P(i-1,j) + u \end{cases} \tag{2.4}$$

$$Q(i,j) = max \begin{cases} M(i,j-1) + w_1 \\ Q(i,j-1) + u \end{cases} \tag{2.5}$$

Apesar do aumento no número de matrizes calculadas, a complexidade de tempo e espaço do algoritmo de Gotoh mantém-se em $O(mn)$ [11].

2.4 Algoritmo Myers-Miller

A complexidade de espaço dos algoritmos vistos nas Seções 2.1, 2.2 e 2.3 inviabilizam a obtenção do alinhamento entre sequências longas. Por exemplo, para alinhar duas sequências de DNA com 1 milhão de nucleotídeos, o algoritmo de Smith-Waterman necessitaria de 4 TB de memória, considerando que cada célula contém 4 *bytes*. Para resolver este problema, algoritmos foram desenvolvidos para obter o alinhamento ótimo utilizando espaço linear, ou seja, na ordem de $O(m+n)$.

A execução da fase 1 dos algoritmos NW, SW e Gotoh (obtenção do escore ótimo) pode ser feita em espaço linear. Entretanto, para uma análise mais detalhada, o significado biológico não é tomado apenas pelo escore, mas sim pelo alinhamento completo.

Hirschberg desenvolveu um algoritmo para recuperar a Maior Subsequência Comum (LCS – *Longest Common Subsequence*) em espaço linear [14]. Este algoritmo foi posteriormente adaptado por Myers e Miller [22] para transformar o algoritmo de Gotoh em uma solução em espaço linear.

A ideia principal do algoritmo Myers-Miller é encontrar o ponto médio pelo qual passa um alinhamento ótimo [22]. Para encontrar este ponto, a computação é feita em duas partes. Na primeira, o processamento é feito por colunas até a coluna central, que corresponde à coluna de número $n/2$. Na segunda parte, o processamento é feito da direita para a esquerda sobre as duas sequências invertidas A_0^r e B_1^r , até que a mesma coluna central $n/2$ seja calculada.

Os valores finais da coluna $n/2$ são armazenados nos vetores CC e DD. A diferença entre os dois vetores é que o vetor DD armazena o escore dos alinhamentos que terminam em *gap* e o vetor CC armazena o escore dos alinhamentos que terminam em *match* ou *mismatch*. Analogamente, o processamento das sequências que são calculadas na ordem inversa são armazenadas nos vetores CC e DD.

Um alinhamento que atravessa a coluna $n/2$ através da linha i terá escore $K_i = min\{CC'_i + CC_i, DD_i + DD'_i - G_{open}\}$, onde G_{open} é a penalidade por abrir um *gap* (i.e.

$G_{open} = G_{first} - G_{ext}$). Observe que, quando somamos o valor dos dois vetores DD e DD', estamos em um caso onde existem *gaps* em ambas as direções, logo precisamos remover a penalidade G_{open} de um dos *gaps*.

O escore K_{i^*} é o valor máximo entre todos os valores de K_i, ou seja, $K_{i^*} = \max_{0 \leq i < m} K_i$. Neste caso, conclui-se que a célula $(i^*, n/2)$ é um ponto pelo qual passa um alinhamento ótimo.

Em seguida, realiza-se recursivamente o processamento de duas seções da matriz: $(0, 0)$ a $(i^*, n/2)$ e $(i^*, n/2)$ a (n, m). A cada passo, dividem-se as seções em trechos menores, até que o tamanho de cada seção seja trivial. A Figura 2.5 ilustra dois passos do algoritmo de Myers e Miller.

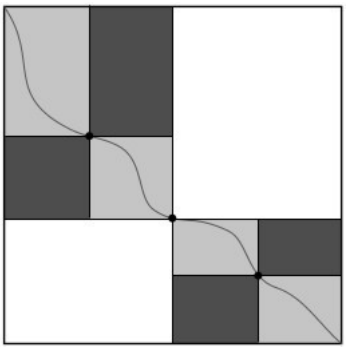


Figura 2.5: Exemplo de alinhamento de sequências utilizando o algoritmo de Myers-Miller na segunda recursão [22].

Haja vista que, a cada passo recursivo, a área de processamento cai pela metade, podemos estimar que o próximo passo gastará metade do tempo do passo anterior. Se T_k é o tempo gasto pelo passo k, então $T_k = T_0/2^k$, sendo $0 \leq k < log_2 m$. Somando todos os tempos é possível conhecer o tempo total, que é $T = \sum_{i=1}^{log_2 m - 1} T_0/2^k \leq 2.T_0$, logo, o algoritmo continua na ordem de $O(mn)$ em tempo, mas com o benefício de utilizar apenas $O(m + n)$ de memória.

2.5 Fickett

Fickett [9] desenvolveu uma otimização capaz de reduzir o tempo de execução e o consumo de memória para obtenção de alinhamento ótimo de sequências para $O(kn)$, onde k é a largura da faixa de diagonais que contém todo o alinhamento ótimo das sequências. Sequências muito similares tendem a produzir alinhamentos próximos da diagonal principal, com poucos *gaps* (isto é, com valores de k pequenos). Logo, o tempo de processamento pode ser consideravelmente reduzido utilizando esta técnica. O algoritmo foi proposto para o problema da distância de edição [13], em que calcula-se uma matriz

d simplificada de maneira parecida com o algoritmo de NW. Entretanto, quanto maior o valor d_{mn} , mais distante (isto é, menos similar) estão as duas sequências.

A principal ideia do algoritmo é reordenar a computação da matriz de programação dinâmica de forma a calcular apenas as células d_{ij} da matriz cujo valor seja menor ou igual a um limitador D. Assim, limita-se o cálculo a uma área da matriz que contem o alinhamento ótimo.

Se o alinhamento não puder ser encontrado com o limite superior D = D_0, escolhe-se um valor $D_1 > D_0$ e continua-se o processamento nas colunas anteriores a K_i e posteriores a L_i [9]. Este procedimento é repetido até que se encontre o alinhamento. A Figura 2.6 ilustra o funcionamento do algoritmo de Fickett.

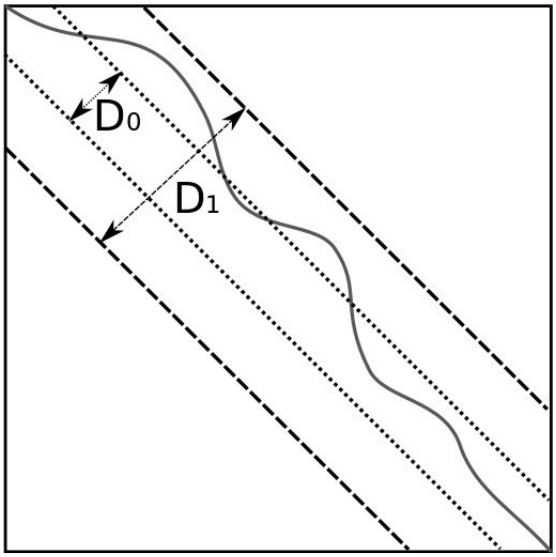


Figura 2.6: Exemplo de alinhamento de sequências utilizando o algoritmo de Fickett com faixas D_0 e D_1 [9].

Em [9] não foi apresentada uma forma de se obter o valor da largura da faixa k precisamente. Ao contrário, são feitas três sugestões:

1. Heurística: o limite superior de D pode ser escolhido, de maneira rápida e não-ótima, por meio de uma heurística. Essa heurística irá utilizar, por exemplo, o valor das pontuações do *match*, *mismatch* e *gap* para determinar a largura da faixa a ser aplicada no alinhamento das sequências. Na Seção 4.2, é proposta uma heurística para a determinação da faixa a ser utilizada;

2. Escolha Manual: o próprio usuário escolhe o valor máximo de D, aproveitando o conhecimento prévio que ele possui das sequências comparadas. Isso pode ser utilizado quando as sequências são sempre comparadas a um grupo de sequências conhecidas previamente, como é o caso da comparação de proteínas com uma base de dados completa de proteínas (i.e.Swiss-Prot);

3. Incremento Automático: um valor inicial D_0 é fixado pelo algoritmo, que calcula as células tais que $d_{ij} \leq D_0$. Caso não seja possível encontrar o alinhamento ótimo com esta restrição, o algoritmo escolhe um valor $D_1 > D_0$ e calcula uma parte maior da matriz tal que $d_{ij} \leq D_1$. O valor D_1 pode ser escolhido, por exemplo, como o dobro do valor D_0. O algoritmo continua gerando novos limites $D_2 > D_3 > . \ . \ .$ até que seja possível encontrar um alinhamento.

Como o alinhamento encontrado depende das pontuações e do grau de similaridade das sequências, não existe uma garantia de que o algoritmo de Fickett possa fornecer um *speedup*, principalmente pelo fato de que determinar o tamanho da faixa é uma tarefa crucial para a aplicação do algoritmo, pois dependendo dos valores escolhidos, é possível até haver uma queda considerável no rendimento, como é o caso de quando se escolhe sucessivamente valores menores do que a largura do alinhamento ótimo das sequências.

Destarte, pode-se observar que os primeiros algoritmos de comparação de sequências biológicas possuiam complexidade de tempo e espaço de ordem quadrática, mas que otimizações foram realizadas para o desenvolvimento de algoritmos com complexidade linear de espaço e, além disso, foram desenvolvidos métodos que reduzem a complexidade temporal ao restringirem a área de computação da matriz de programação dinâmica.

Capítulo 3

Comparação de Sequências Biológicas em Plataformas de Alto Desempenho

Como visto no Capítulo 2, a comparação de sequências biológicas com algoritmos exatos possui um alto custo computacional. Ao desenvolver algoritmos para esse problema, cientistas se esforçaram durante anos para extrair o máximo da capacidade computacional disponível para que a resolução fosse feita em tempo viável. Na Seção 3.1 são apresentadas algumas ferramentas que usam CPUs e *clusters* de CPUs na comparação de sequências. Na Seção 3.2, serão apresentados conceitos referentes a GPUs e algumas ferramentas que usam GPU na comparação de sequências. Finalmente, a Seção 3.3 apresenta um comparativo entre as ferramentas.

3.1 Comparação de Sequências em CPU

As CPUs são utilizadas em grande parte do processamento que é realizado nos computadores, mesmo em computadores que possuem aceleradores como as GPUs. Na comparação de sequências biológicas não é diferente, a CPU é um dos principais processadores das soluções desenvolvidas. Até o ano 1997, nenhuma ferramenta de alinhamento de sequências biológicas utilizava as instruções vetoriais das CPUs, mas a partir de então seu uso foi difundido cada vez mais. Nas Seções 3.1.1 a 3.1.4 serão apresentadas algumas ferramentas de comparação de sequências em CPU.

3.1.1 Rajko e Aluru (2004)

Em [26] é proposto um algoritmo para alinhamento global de sequências usando o modelo *affine gap*. Este algoritmo combina o algoritmo de Myers-Miller (Seção 2.4) com o algoritmo de computação em prefixo paralelo. A computação com prefixo paralelo pode

ser utilizada para quebrar a dependência entre *threads* vizinhas, permitindo que todas as *threads* comecem a computar a mesma linha paralelamente. Para isso, a matriz de programação dinâmica é dividida em n/t partes, onde t é o número de *threads*. A cada iteração i, t *threads* calculam suas células em paralelo com uma equação que computa o máximo prefixo local. Na proposta original de prefixo paralelo, após o cálculo de uma iteração, as *threads* trocam seus valores em tempo $O(\log t)$ de maneira a aplicar um fator de correção nos resultados.

Para reduzir a comunicação e aumentar o paralelismo, os autores de [26] particionaram a matriz de programação dinâmica entre as *threads* de maneira que o alinhamento global ótimo seja a concatenação dos alinhametos encontrados em cada subproblema. Assim, o algoritmo proposto por [26] é capaz de obter um particionamento entre as subsequências das sequências m e n que estão sendo comparadas e com este particionamento, o problema pode ser dividido em subproblemas independentes, que podem ser resolvidos paralelamente, otimizando os recursos computacionais que estão disponíveis e aumentando o paralelismo.

Este algoritmo foi testado em um *cluster* com 60 processadores. Uma taxa de 0.24 GCUPS (Bilhões de Células Atualizadas por Segundo) foi obtida quando duas sequências de tamanho 1.100.000 foram comparadas.

3.1.2 Batista et Al (2008)

O algoritmo paralelo Z-align é descrito em [2]. O Z-align é uma estratégia paralela baseada em MPI derivada do método *block-based wavefront* para executar o algoritmo Gotoh (Seção 2.3) combinado com Fickett (Seção 2.5). Duas matrizes de programação dinâmica adicionais D_{inf} e D_{sup} (chamadas de matrizes de divergência) são propostas, com o objetivo de restringir o espaço no procedimento de *traceback*, em uma versão adaptada do algoritmo de Fickett (Seção 2.5). As matrizes adicionais são responsáveis por armazenar o valor da faixa de Fickett para o alinhamento de prefixos representado por cada célula da matriz. Sendo assim, ao se terminar a fase 1 do algoritmo Gotoh e obter-se a célula que contém o maior escore, basta consultar D_{inf} e D_{sup} para aquela célula, de modo a obter-se os valores inferior e superior da faixa de Fickett. A fase de *backtracking* é feita, então, utilizando-se essa faixa.

O Z-align foi capaz de comparar sequências de até 3 milhões de pares de bases. O melhor GCUP (0.21) foi obtido em um *cluster* de 8 nós com dois núcleos cada, na comparação de sequências de DNA reais de tamanho 3.147.090 e 3.282.708.

Em [3], este trabalho foi estendido para computar a matriz de programação dinâmica de maneira cíclica, dividindo-a em xp subconjuntos de colunas, onde p é o número de unidades de processamento e x é uma constante. Uma taxa de 1.39 GCUPS foi obtida na

comparação de sequências reais de DNA de tamanho 23MBP (Milhões de Pares de Bases) x 24MBP em um *cluster* de 64 núcleos.

3.1.3 Rognes e Seeberg (2000) - SWMMX

Em [27] foi proposta uma implementação do algoritmo de Gotoh (Seção 2.3) que compara sequências de proteínas com *affine gap* usando as instruções vetoriais MMX/SSE da Intel. As células são calculadas em paralelo, coluna por coluna, aplicando a otimização SWAT. A otimização SWAT foi proposta por [12] e é aplicada ao modelo *affine gap*. Esta técnica é baseada na observação que as matrizes que tratam as sequências de *gaps* geralmente contém células com valor zero, as quais não contribuem para a computação da matriz do escore. Com isso, a matriz principal somente precisa acessar a matriz dos *gaps* quando o valor correspondente for maior do que a penalidade para abrir e estender um *gap*. Quebrando a dependência da matriz P (Equação (2.4)), a computação de cada coluna da matriz principal pode ser feita paralelamente e a matriz P dos *gaps* é calculada apenas quando isso é necessário, economizando tempo de processamento. Essa otimização é muito efetiva quando a penalidade de abertura de *gap* não é muito pequena.

Além de usar a otimização SWAT, Rognes e Seeberg propuseram a otimização *query profile* [27], que computa uma pequena matriz específica baseada na *query sequence* e numa matriz de substituição dada (Seção 2.1). Essa matriz é utilizada mais tarde na computação da matriz de programação dinâmica, ao invés da matriz de substituição. Cada instrução MMX/SSE age em 8 células (i até $i + 8$) em uma única coluna da matriz de programação dinâmica, calculando essas células no modo SIMD (*Single Instruction Multiple Data*) paralelo. Os valores dos escores são restritos a 8 bits (0 a 255).

Experimentos foram realizados comparando 11 sequências, com tamanhos variando de 189 a 567 aminoácidos, usando a base de dados genômica Swiss-Prot [1]. Foram atingidos 0.15 GCUPS com um Pentium III 500MHz.

3.1.4 Farrar (2007)

O algoritmo proposto em [8] usa o conceito de paralelismo baseado em células e o *query profile* discutido na Seção 3.1.3, mas com um padrão diferente de acesso aos dados, chamado *striped*. No padrão *striped*, as células que são calculadas com instruções vetoriais pertencem a uma mesma coluna e são, por exemplo, 1, 1 + t, 1 + 2t, 1 + 3t, com um fator *stripe* de t. Esse padrão *striped* reduz as dependências de dados, movendo o teste da matriz de *gaps* para o laço de repetição externo, aumentando o paralelismo e acelerando a computação.

O algoritmo Farrar foi implementado em C, usando instruções SSE2, e os testes foram realizados em um Intel Xeon Core 2 Duo 2.0GHz. Sequências com tamanho de até 567 aminoácidos foram comparadas a toda a base de dados genômica Swiss-Prot [1], e um pico de desempenho de 3.0 GCUPS foi atingido.

3.2 Comparação de Sequências em GPU

As unidades de processamento gráfico (GPUs) são utilizadas para a comparação de sequências biológicas devido ao alto grau de paralelismo que essas permitem [6]. Na Seção 3.2.1 será apresentada a definição e a arquitetura das GPUs. Na Seção 3.2.2 será descrita a ferramenta CUDASW++. Na Seção 3.2.3 a ferramenta SW# é apresentada. Na Seção 3.2.4, a ferramenta CUDAlign é descrita em detalhes, pois a mesma foi utilizada na implementação da nossa proposta.

3.2.1 GPUs

Uma das plataformas de alto desempenho que vem sendo muito utilizada em comparação de sequências é a GPU (*Graphics Processing Unit*). As unidades de processamento gráfico (GPUs) foram desenvolvidas para acelerar jogos e processamento de imagens, porém sua estrutura de alto paralelismo faz delas mais efetivas do que as CPUs para uma série de outros algoritmos, incluindo algoritmos de comparação de sequências.

Uma CPU tradicional, como um processador da arquitetura Intel® Haswell, atinge no máximo 177 GigaFLOPS (Bilhões de Operações de Ponto Flutuante por Segundo) em precisão simples, enquanto uma GPU GTX TITAN Z pode atingir 8,1 TeraFLOPS em precisão simples [24].

Uma das razões para a lacuna de desempenho ser muito grande entre CPU com múltiplos núcleos (*multicore*) e GPU com muitos núcleos (*many-core*) está na filosofia de projeto empregada nos dois tipos de unidades de processamento: a GPU ocupa a maior parte da sua área com unidades de processamento enquanto a CPU gasta uma área considerável com memória cache e unidades de controle [24].

As GPUs foram concebidas para processar grande conjunto de dados, por exemplo, na renderização gráfica, que atua sobre dados em posições contínuas na memória, caracterizando um vetor ou *array* de dados. Por outro lado, no processamento de dados que não envolva gráficos, as operações requisitadas não se realizam predominantemente sobre posições contínuas da memória. A cada ciclo de *clock*, as operações são realizadas sobre dados diferentes, que neste caso, podem estar armazenados em posições não contínuas da memória. Este tipo de processamento é chamado de escalar, e os processadores para este

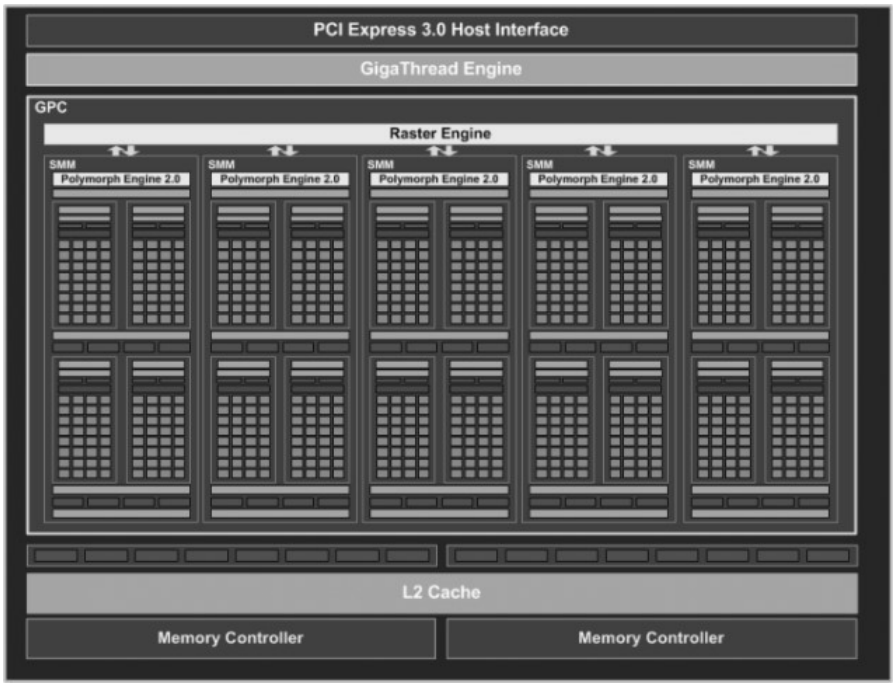


Figura 3.1: Arquitetura Maxwell em diagrama de blocos de alto nível [24].

fim, são chamados "processadores escalares", como as tradicionais CPUs. Na taxonomia de Flynn [10] diz-se que são SISD (*Single Instruction Single Data*).

As GPUs se distinguem por sua grande capacidade de executar cálculos simultâneos sobre um conjunto de dados. Pela taxonomia de Flynn, as GPUs possuem características SIMD (*Single Instruction Multiple Data*) e, por serem concebidas para operarem principalmente sobre vetores, elas podem ser vistas como "processadores vetoriais".

O estado da arte em projeto de GPU é representado pela nova geração da Nvidia, de codinome Maxwell. A Figura 3.1 mostra um diagrama de blocos de alto nível da primeira placa de vídeo da arquitetura Maxwell. Como pode ser visto, essas GPUs são compostas por vários *Streaming Multiprocessors* (SMM) que compartilham uma cache L2 e uma memória global. As GPUs se ligam ao *host* através do barramento PCI. A Figura 3.2 mostra, em detalhe, um SMM.

Cada SMM é composto é composto por 4 unidades, contendo cada uma 32 núcleos, com uma cache L1/textura e uma memória compartilhada. Os núcleos do SMM operam segundo o modelo SIMT (*Single Instruction Multiple Thread*) onde todas as *threads* executam a mesma instrução, sobre dados diferentes.

As GPUs NVidia [25] são geralmente programadas com CUDA (*Compute Unified Device Architecture*). CUDA é a arquitetura de hardware e de software da NVidia [25] para programação em GPGPU (*General Purpose Graphics Processing Unit*) através de linguagens de programação como C, C++ e Fortran e de frameworks como OpenCL e DirectCompute. Esta arquitetura inclui um conjunto de instruções (ISA), uma plataforma

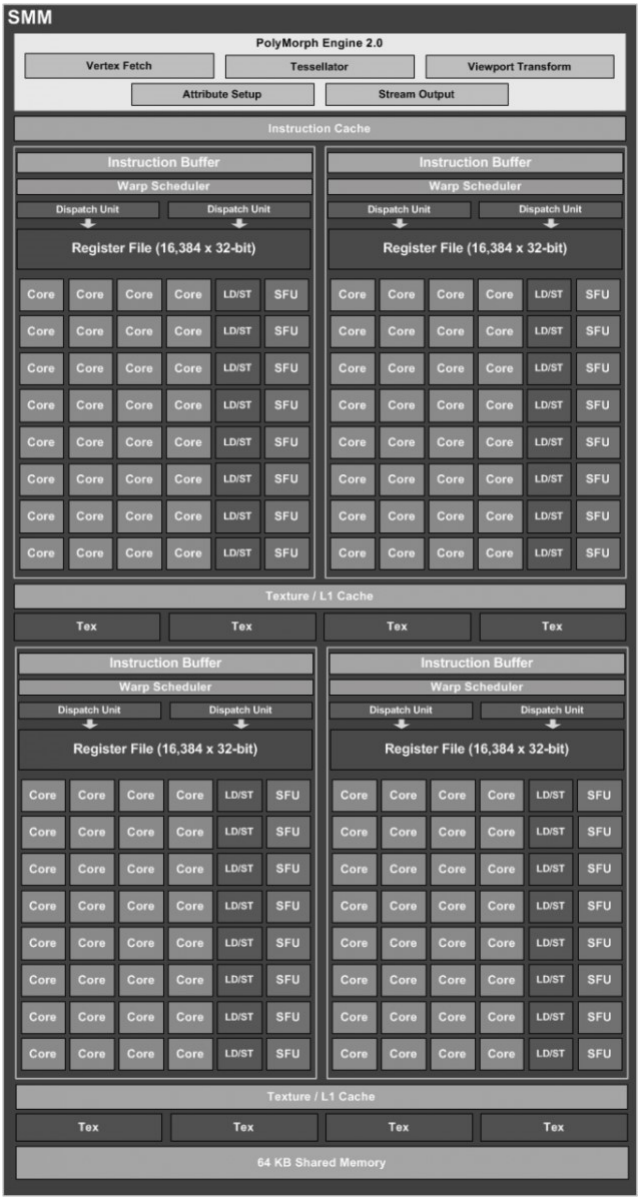


Figura 3.2: Multiprocessadores de *streaming* específicos da arquitetura Maxwell (SMM). [24]

de desenvolvimento (SDK) e o próprio hardware em GPU.

A seguir, apresentaremos três ferramentas que usam GPUs para a comparação de sequências.

3.2.2 CUDASW++

A ferramenta CUDASW++ foi proposta em [18] para a comparação de proteínas em GPU com o algoritmo Gotoh (Seção 2.3). O CUDASW++ se encontra atualmente em sua terceira versão. Nas seções a seguir será descrita a ferramenta CUDASW++ em suas diferentes versões.

CUDASW++1.0

O CUDASW++1.0 [18] executa o algoritmo Gotoh (Seção 2.3) em GPUs para comparações de proteínas. Esta proposta divide o problema em várias tarefas, sendo que cada tarefa compara a sequência de busca com uma das sequências do banco genômico. Sendo assim, foram propostos dois métodos para paralelizar o problema: paralelização inter-tarefas e paralelização intra-tarefas.

Na abordagem inter-tarefas, cada comparação é atribuída e processada por uma única *thread*. Executam-se blocos com 256 *threads*, sendo que a quantidade de blocos executada em paralelo é igual ao número de multiprocessadores (SMM) da GPU. Já na abordagem intra-tarefas, cada comparação é atribuída e processada por um bloco de *threads*, sendo que todas as 256 *threads* deste bloco cooperam para processar a mesma comparação. O paralelismo é obtido por meio da técnica de *wavefront*, em que as células das anti-diagonais são processadas em paralelo. Para tratar o problema das anti-diagonais possuírem um número distinto de células, a implementação divide as anti-diagonais em blocos de tamanhos fixos e considera terem virtualmente o mesmo número de células [18].

Segundo os resultados experimentais, a abordagem inter-tarefas é mais rápida que a intra-tarefas, mas a intra-tarefas permite comparar sequências maiores. Sendo assim, a execução da comparação é dividida em duas fases, onde a primeira fase utiliza o método inter-tarefas para comparar sequências menores que um determinado tamanho e a segunda fase compara as demais sequências com a abordagem intra-tarefas.

Com o CUDASW++1.0, foram realizados testes na GPU NVidia GeForce GTX 280 e na GPU NVidia GeForce GTX 295, proporcionando um significante ganho de desempenho comparado a outras implementações. O CUDASW++1.0 suporta sequências de busca de tamanho de até 59K e, nos testes, comparou 25 sequências de busca variando do tamanho 144 a 5.478 com o banco genômico *Swiss-Prot* [1] versão 56.6. O CUDASW++1.0

alcançou um desempenho máximo de 9,66 GCUPS na GPU NVidia Geforce GTX 280 e o desempenho máximo de 16,087 GCUPS na GPU Nvidia Geforce GTX 295.

CUDASW++2.0

O CUDASW++2.0 [19] propõe otimizações para acelerar ainda mais a comparação de sequências. No CUDASW++2.0, duas implementações do Gotoh (Seção 2.3) foram propostas. A primeira é uma otimização da versão inter-tarefas que, nesta nova versão, utiliza duas otimizações: *sequential query profile* (Seção 3.1.3) e *packed data format*. A segunda implementação é uma variante do padrão *striped* proposto por Farrar (Seção 3.1.4).

O *sequential query profile* é armazenado na memória de textura, substituindo o acesso aleatório à matriz de substituição na memória compartilhada. Deste modo, quatro pontuações de substituição são computadas utilizando apenas uma busca de textura, melhorando significativamente a transferência de memória de textura. Como no *query profile*, uma das sequências a serem comparadas (sequência *query*) é também reorganizada usando o *packed data format*, onde quatro resíduos sucessivos de cada sequência *query* são empacotados e representados usando o tipo de dados vetorial. Neste caso, quando se utiliza o método de divisão bloco de células, os quatro resíduos carregados para uma busca de textura são subsequentemente armazenados em memória compartilhada para a utilização do laço interno do código proposto.

O CUDASW++2.0 atingiu uma melhoria do desempenho sobre o CUDASW++1.0 de 1,74 (1,72) vezes, utilizando a primeira implementação, ou superior a 1,77 (1,66) vezes usando a segunda implementação, com um desempenho de até 17 (30) GCUPS em uma GPU GeForce GTX 280 (dupla-GPU GeForce GTX 295).

CUDASW++3.0

Em [20] é apresentada a versão 3.0 do CUDASW++, um algoritmo no qual instruções SIMD de CPU e GPU realizam computações simultâneas nas duas plataformas. Sendo assim, a versão 3.0 do CUDASW++ é uma das primeiras ferramentas a utilizar uma arquitetura de computação híbrida (CPU+GPU).

Para a computação em GPU foi utilizada uma paralelização SIMD que utiliza instruções de vídeo de baixo nível CUDA PTX para ganhar maior paralelismo de dados, além do modelo de execução SIMT (*Single Instruction Multiple Threads*). Na CPU, instruções SSE de processamento SIMD são usadas. Além do mais, cargas de trabalho são automaticamente distribuídas sobre as CPUs e GPUs com base em suas respectivas capacidades de computação.

Testes no banco de dados Swiss-Prot [1] mostram que o CUDASW++3.0 obtém uma melhoria de desempenho sobre o CUDASW++2.0 até 2.9 e 3.2 vezes, com um máximo

desempenho de 119,0 e 185,6 GCUPS, em um Intel® i7 2700K *quad-core* 3.5 GHz e uma placa de vídeo de uma única GPU GeForce GTX 680 e uma dupla-GPU GeForce GTX 690, respectivamente.

3.2.3 SW#

No artigo de Korpar e Sikic [17] é apresentada a implementação do algoritmo de Myers-Miller (Seção 2.4) no SW#. O SW# (*swsharp*) é uma ferramenta para alinhamento de sequências baseado em GPUs que utilizam a linguagem CUDA [17]. Essa ferramenta é capaz de utilizar até duas GPUs para acelerar a computação da matriz.

Na ferramenta SW# existem três módulos:

SW#n: Módulo que é utilizado para comparar cadeias de nucleotídeos. Esse módulo é otimizado para sequências longas, usando otimizações propostas no CUDAlign (Seção 3.2.4) para o cálculo da matriz de programação dinâmica.

SW#p: Módulo que tem por finalidade o alinhamento de proteínas.

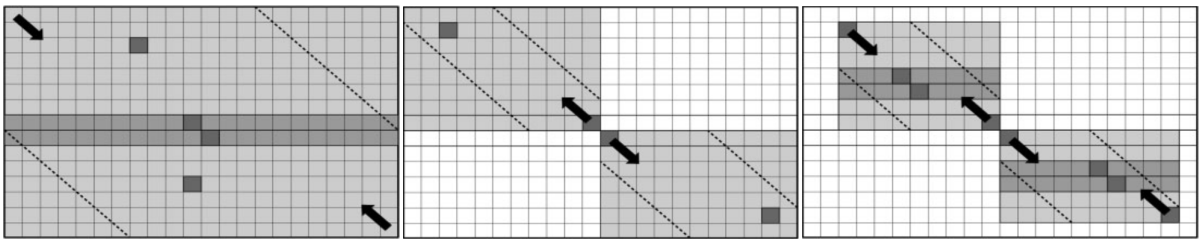
SW#db: Módulo desenvolvido para comparar uma cadeia de nucleotídeos ou de proteínas a uma base de dados. Nessa opção é possível escolher uma base de dados existente ou definir uma base de dados personalizada.

A solução SW# [17] é desenhada em três fases: resolução, localização e reconstrução. Na fase de resolução, são utilizadas as mesmas técnicas que serão apresentadas no CUDAlign 2.1 (Seção 3.2.4) para execução do algoritmo de Myers-Miller (Seção 2.4) na comparação de sequências longas de DNA em plataforma CUDA (Figura 3.3a). Como otimização, os autores sugerem que essa fase divida o problema inicial em dois subproblemas depois da execução da primeira fase do algoritmo de Myers-Miller, após obtidos o ponto e o escore médios da matriz, pois de posse dos *crosspoints* da linha do meio é possível buscar em paralelo a localização do ponto de início da parte superior e o ponto de início do reverso da parte inferior. Isto permite que o problema possa ser distribuído para duas placas gráficas nesta etapa da execução (como pode ser visto na Figura 3.3b), acelerando a localização do ponto final do alinhamento.

Na fase de localização, o algoritmo de Myers-Miller é executado de maneira reversa para localizar o ponto de partida do alinhamento.

O método de *wavefront* é utilizado na fase de reconstrução, combinado com a execução recursiva do algoritmo de Myers-Miller modificado para que a execução seja interrompida quando o tamanho de uma submatriz é inferior ao limite definido. Neste ponto, a execução

é passada para a CPU, responsável por executar a tarefa de *traceback* para gerar o alinhamento obtido nesta iteração, que será combinado aos demais para gerar o alinhamento completo (Figura 3.3c). Uma representação básica do funcionamento das três etapas pode ser vista na Figura 3.3, sendo: (a) resolução, (b) localização, e (c) reconstrução.



(a) Resolução. (b) Localização. (c) Reconstrução.

Figura 3.3: Fases da solução SW# [17].

Sequências de até 47 MBP foram alinhadas nos experimentos. Os resultados obtidos foram semelhantes ao CUDAlign 2.1 em uma única placa gráfica, sendo até mesmo mais lento nos testes com sequências mais longas. A melhoria de desempenho (medido em tempo de execução) foi observada apenas com a utilização da placa NVidia Geforce GTX 690 dual. Baseando-se no tamanho da sequência testada e no tempo de execução, calcula-se que a solução obteve 65,2 GCUPS. Como vantagem em relação ao CUDAlign, esta abordagem não necessita de espaço em disco adicional para armazenamento de linhas/colunas especiais para produzir um alinhamento ótimo.

3.2.4 CUDAlign

O CUDAlign é uma ferramenta proposta em [28] e que já possui várias versões após a concepção original. Nas próximas seções será explicado o CUDAlign e o que foi acrescentado de funcionalidade em cada uma das novas versões criadas.

CUDAlign1.0

Em [28] é proposta a ferramenta denominada CUDAlign 1.0. A finalidade da ferramenta é efetuar comparações de sequências de DNA que possuam mais de 1 MPB (milhão de pares de base). Para realizar a comparação é utilizado o algoritmo de Gotoh (Seção 2.3) e a saída do programa é o escore ótimo.

A implementação do CUDAlign 1.0 utiliza estruturas de memória de ordem linear $O(m + n)$, levando em consideração a comparação de uma sequência de tamanho m e outra de tamanho n.

O CUDAlign 1.0 foi projetado utilizando a técnica de *wavefront* para obter paralelismo. Essa técnica consiste em processar iterativamente cada anti-diagonal da matriz, de forma que cada célula da mesma anti-diagonal possa ser processada paralelamente. O mesmo conceito é aplicado agrupando células em blocos, formando anti-diagonais de blocos que também são processadas paralelamente.

No CUDAlign 1.0, a técnica de *wavefront* foi aplicada em dois níveis: paralelismo externo e paralelismo interno. O primeiro nível de paralelismo ocorre entre blocos de células, e o outro nível de paralelismo ocorre no interior de um bloco. Dessa forma, o CUDAlign divide as matrizes em setores e subsetores com o objetivo de paralelizar as comparações de sequências e empregar o *wavefront*. Na Figura 3.4 é caracterizada a divisão de blocos no CUDAlign, e na Figura 3.5 a divisão interna dos blocos é explanada.

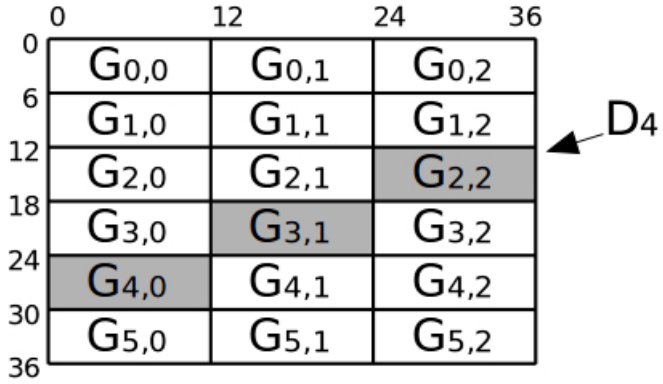


Figura 3.4: Divisão da matriz em blocos [28]. A diagonal externa D_4 está com blocos em destaque.

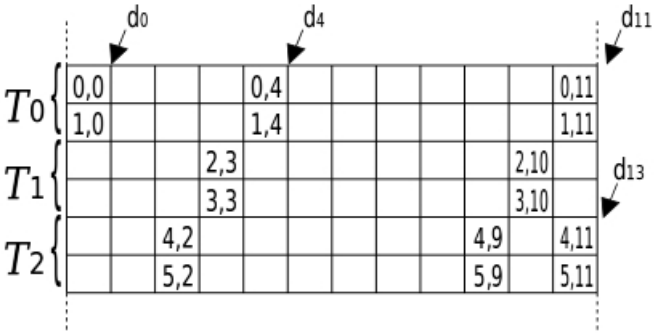


Figura 3.5: Divisão em diagonais internas em bloco retangular com 3 *threads*. Cada *thread* é responsável por 2 linhas do bloco. As diagonais internas d_0, d_4, d_{11} e d_{13} estão indicadas na Figura [28].

De modo a explorar o paralelismo *wavefront*, os blocos do CUDAlign não são retangulares e sim possuem a forma de paralelogramos, como mostrado na Figura 3.6. Esses blocos são processados em GPU com uma técnica chamada "*cells delegation*" [28].

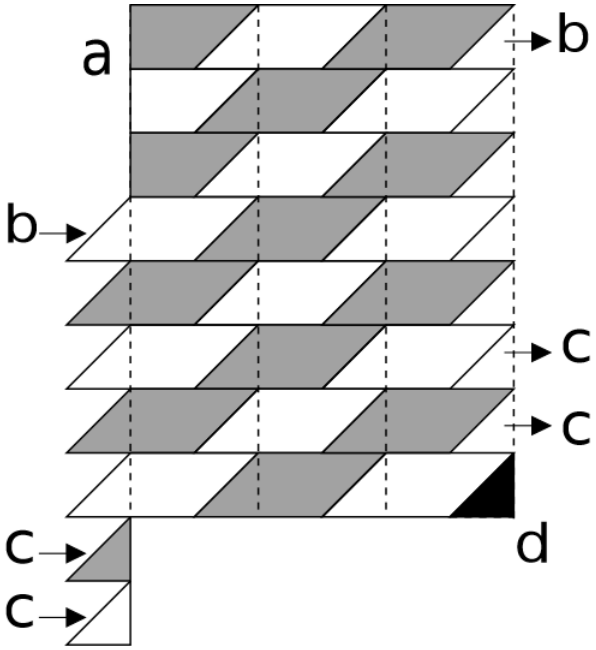


Figura 3.6: Aplicação da Técnica *Cells Delegation* [28].

Para que o CUDAlign 1.0 efetue o processamento de sequências longas de DNA, foram desenvolvidas estruturas de dados otimizadas para os tipos de memórias existentes na arquitetura CUDA [28], atentando para o fato de que na matriz de programação dinâmica cada célula possui três componentes: M, P e Q (Seção 2.3), sendo que cada componente possui 32 *bits* (4 *bytes*), que corresponde ao espaço alocado na memória para um número inteiro. As estruturas de dados são definidas nos seguintes tipos:

Sequências de Entrada: As sequências de entrada são armazenadas em textura (memória somente-leitura). Cada nucleotídeo ocupa um *byte* em memória.

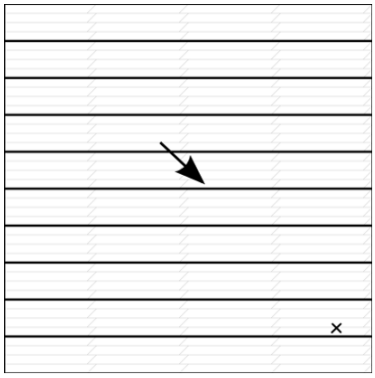
Memória Compartilhada: Os valores da última linha de cada *thread* são transferidos através de memória compartilhada, salvo no caso da última *thread*, que envia os valores através da memória global para a primeira *thread* do próximo bloco. Os componentes que precisam ser transferidos entre as *threads* são apenas os componentes M e P.

Barramento Horizontal: Os valores da última linha de cada bloco são armazenados nesta área da memória global para que o bloco imediatamente inferior carregue esses dados e continue o seu processamento. As operações de escrita são feitas diretamente na memória global, mas as leituras são feitas através de textura.

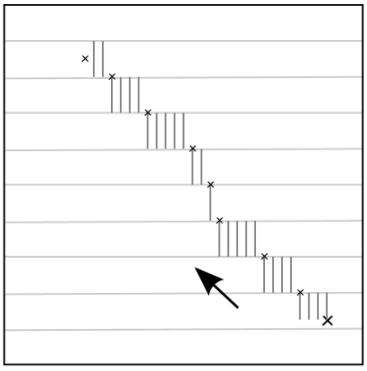
Resultados foram coletados na GPU NVidia GTX 280, comparando sequências de até 32 MBP, alcançando um máximo de 20,37 GCUPS.

CUDAlign2.0

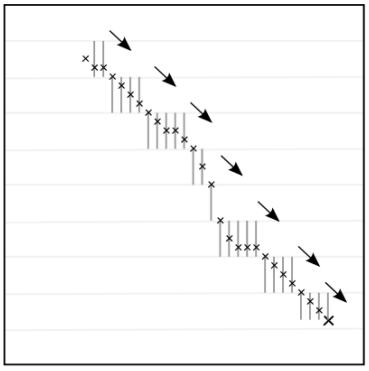
O CUDAlign 2.0 [7] tem por objetivo recuperar o alinhamento de sequências longas com uma GPU. Para tanto, foi dividido em seis estágios (Figura 3.7). O estágio 1 executa a fase 1 do algoritmo Gotoh. Os estágios 2 a 5 executam a fase 2 (*backtracking*). O estágio 6 é opcional, usado para visualização do alinhamento. Em seguida cada estágio será detalhado.



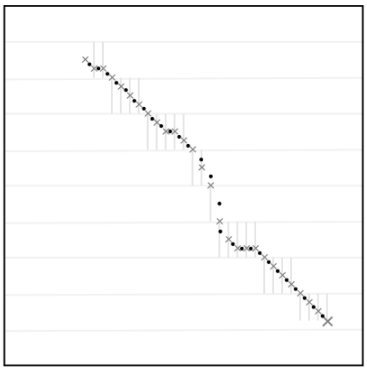
(a) O Estágio 1 encontra o escore ótimo e sua posição. Linhas especiais são salvas em disco.



(b) O Estágio 2 encontra as coordenadas nas quais o alinhamento ótimo intercepta as linhas especiais. Colunas especiais são salvas em disco.

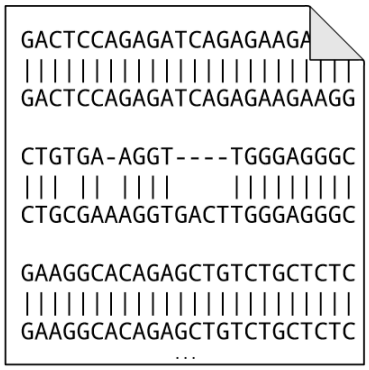


(c) O Estágio 3 encontra as coordenadas que interceptam o alinhamento ótimo pelas colunas especiais salvas no Estágio 2.



(d) O Estágio 4 executa o algoritmo de Myers e Miller em cada partição formada por coordenadas sucessivas.

(e) O Estágio 5 encontra as coordenadas que interceptam o alinhamento ótimo pelas colunas especiais salvas no Estágio 2.



(f) O Estágio 6 executa o algoritmo de Myers e Miller em cada partição formada por coordenadas sucessivas.

Figura 3.7: Estágios do CUDAlign [30].

Estágio 1 O objetivo do Estágio 1 é encontrar o escore ótimo e a coordenada final do alinhamento. O Estágio 1 calcula o escore ótimo utilizando o modelo de *affine-gap* (Seção 2.3). O Estágio 1 executa o algoritmo básico do CUDAlign 1.0 com uma modificação: algumas linhas da matriz (chamadas linhas especiais) são salvas em disco. As linhas especiais são utilizadas para executar o procedimento de *matching* no estágio 2 e para efetuar *checkpoints*, permitindo que a execução seja posteriormente restaurada em caso de interrupção na execução do algoritmo.

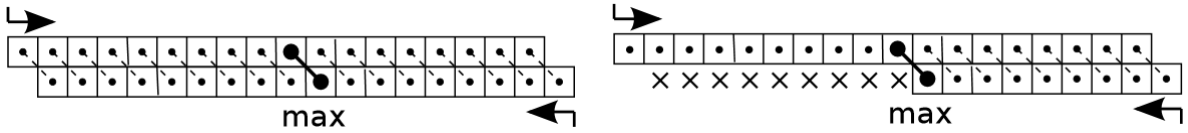
As linhas especiais são obtidas do barramento horizontal a cada intervalo de x linhas (linhas em *bold* na Figura 3.7a).

A Figura 3.7a apresenta as saídas do Estágio 1. Neste exemplo, o intervalo entre as linhas especiais é de 4 linhas. A posição do escore ótimo está representada por um "✕".

Estágio 2 O objetivo do Estágio 2 (Figura 3.7b) é encontrar as coordenadas de um alinhamento ótimo (*crosspoints*) que cruzam as linhas especiais salvas no Estágio 1. Além disso, o Estágio 2 deve encontrar a coordenada inicial do alinhamento ótimo. As saídas do Estágio 2 estão ilustradas na Figura 3.7b. Duas otimizações foram propostas para esse estágio: 1) procedimento de *matching* baseado em objetivo e 2) execução ortogonal.

***Matching* baseado em objetivo:** O algoritmo Myers-Miller original (Seção 2.4) executa um procedimento de *matching* para encontrar a célula da linha central através da qual o alinhamento ótimo passa. Nesse procedimento, todas as células da linha central são analisadas, de forma a encontrar o par de células cujo escore somado seja o maior possível. No estágio 2, diferentemente do algoritmo Myers-Miller original, o escore máximo já é conhecido e este é chamado de escore-alvo. Logo, ao contrário de Myers-Miller, assim que o escore-alvo for encontrado, o procedimento de *matching* pode encerrar a sua execução e iniciar uma nova iteração em busca da próxima coordenada (relativa à próxima linha especial). Inicialmente, o escore-alvo é o próprio escore ótimo obtido no Estágio 1, mas à medida que novas coordenadas são encontradas, o escore-alvo é atualizado com o escore da última coordenada obtida da linha especial. A Figura 3.8 ilustra a diferença entre o procedimento de *matching* do algoritmo Myers-Miller original e o procedimento de *matching* baseado em objetivo.

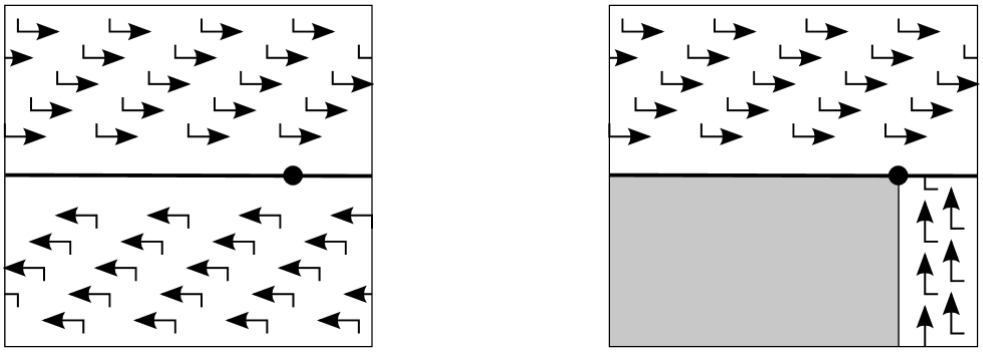
Execução ortogonal: Para haver ganho de desempenho ao utilizar o procedimento de *matching* baseado em objetivo, as *threads* do Estágio 2 não podem executar na mesma direção horizontal do Estágio 1. Em vez disso, a execução das *threads* deve ser feita verticalmente, em direção ortogonal ao Estágio 1 (Figura 3.9a). Desta forma, diminui-se a área processada até que o procedimento de *matching* ocorra. A Figura 3.9b ilustra a redução da área processada. Para reduzir ainda mais a área processada durante o procedimento de *matching* ortogonal, foi proposta uma otimização nesse procedimento,



(a) Procedimento original. (b) Procedimento baseado em objetivo.

Figura 3.8: Procedimentos de *Matching*. O procedimento original de *matching* do algoritmo Myers-Miller (a) compara todos os pares de células para encontrar onde ocorre o escore máximo (*max*). No procedimento de *matching* baseado em objetivo (b), o maior escore já é conhecido, então o procedimento encerra assim que o escore máximo (*max*) for encontrado. O × indica as células que não foram processadas [7].

fazendo com que as duas partes do bloco sejam processadas na direção da linha do meio, gerando um ganho de desempenho ainda maior. A Figura 3.12 exemplifica essa otimização.

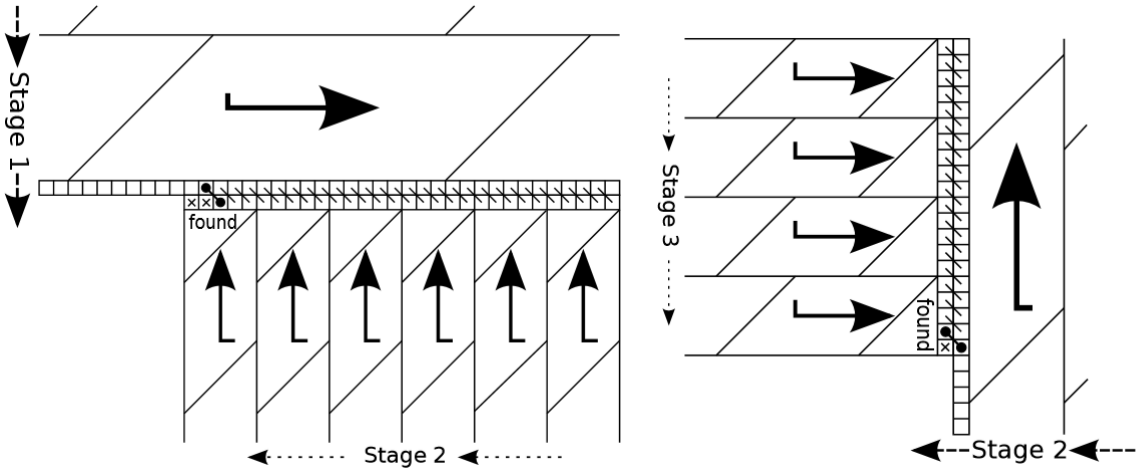


(a) Execução original. (b) Execução ortogonal.

Figura 3.9: Execução ortogonal. A execução original do algoritmo Myers-Miller (a) processa as duas metades da matriz na mesma direção horizontal. Utilizando a execução ortogonal (b), o processamento da parte inferior é feito na vertical, em direção ortogonal ao da parte superior. A área em cinza não precisa ser processada, pois o escore-alvo foi encontrado no círculo preto [7].

Estágio 3 O objetivo do Estágio 3 é aumentar o número de coordenadas conhecidas que cruzam o alinhamento ótimo. O Estágio 3 é praticamente idêntico ao Estágio 2. A diferença entre ambos é que, no Estágio 3, as coordenadas sucessivas formam partições com começo e fim bem definidos, ao contrário do Estágio 2, que conhece a coordenada final e a coordenada inicial precisa ser encontrada.

Visto que existem partições bem definidas no Estágio 3, executa-se o mesmo algoritmo básico em GPU para encontrar as coordenadas por onde o alinhamento ótimo intercepta as linhas especiais salvas no Estágio 2. Note que não existe dependência entre as partições e, por isso, a ordem de execução das partições é irrelevante. Sendo assim, as partições podem ser processadas em paralelo ou até em GPUs distintas. Cada partição é dividida em várias



(a) Estágio 1 e Estágio 2. (b) Estágio 2 e Estágio 3.

Figura 3.10: Detalhe no procedimento de *matching* ortogonal [7].

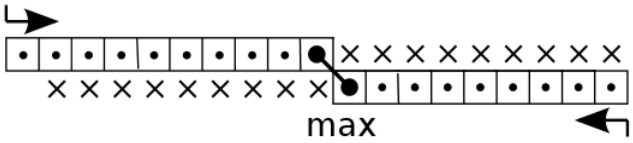


Figura 3.11: Detalhe no procedimento de *matching* ortogonal otimizado.

coordenadas, uma para cada intersecção entre o alinhamento ótimo e as linhas especiais salvas no Estágio 2 usando o *matching* por objetivo e a execução ortogonal (Figura 3.10). A Figura 3.7c apresenta as saídas do Estágio 3 e a Figura 3.10b apresenta, em detalhe, uma região onde a coordenada do alinhamento foi encontrada durante o procedimento de *matching*.

Estágio 4 O Estágio 4 executa em CPU o algoritmo de Myers-Miller (Seção 2.4) em cada partição formada por coordenadas sucessivas encontradas ao final do Estágio 3. O objetivo do Estágio 4 (Figura 3.7d) é aumentar, iterativamente, o número de coordenadas do alinhamento ótimo conhecidas até que o tamanho de cada partição formada entre as coordenadas seja menor que uma constante chamada de *tamanho máximo de partição*. O tamanho máximo de partição foi escolhido empiricamente como sendo 16 bases, de forma que o estágio 5 obtivesse o alinhamento destas pequenas partições em poucos segundos.

Cada iteração do Estágio 4 pode aumentar até duas vezes o número de coordenadas conhecidas. Por isso, várias iterações podem ser necessárias até que os tamanhos das partições sejam inferiores ao tamanho máximo de partição. Assim como no Estágio 3, a ordem de processamento das partições é irrelevante, então as partições são processadas por várias *threads* em paralelo.

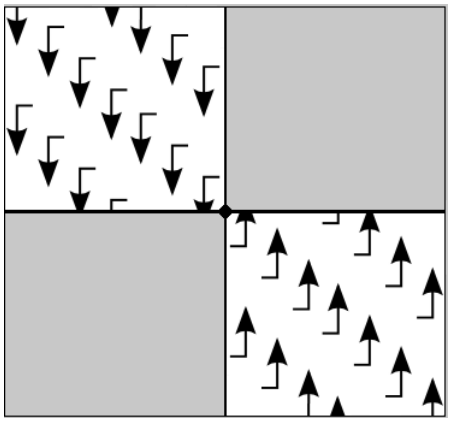


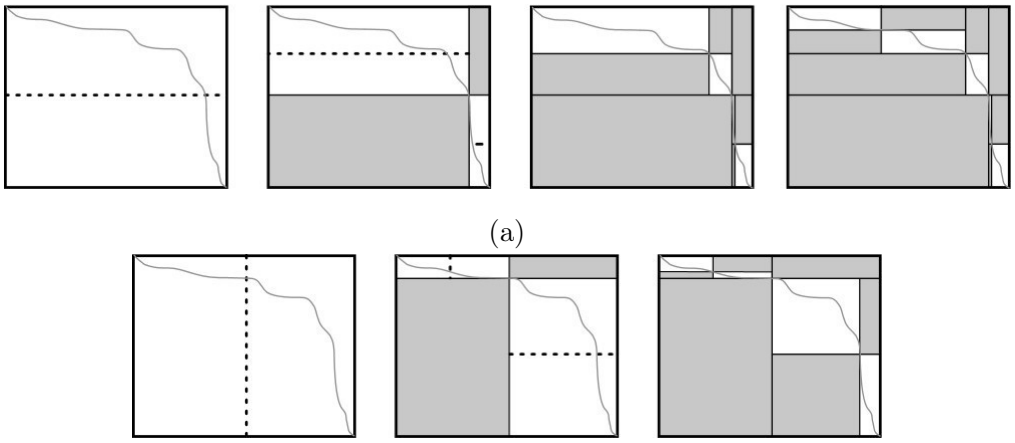
Figura 3.12: Execução ortogonal otimizada. A área em cinza não precisa ser processada, pois o escore-alvo foi encontrado no círculo preto.

Conforme visto na Seção 2.4, o algoritmo Myers-Miller divide a matriz em sua linha central e processa as duas metades da matriz em sua totalidade e em sentido contrário. Por fim, as últimas linhas de cada uma das metades da matriz são comparadas de forma a encontrar a coordenada através da qual o alinhamento ótimo passa. No Estágio 4, o algoritmo Myers-Miller foi otimizado com uma proposta de *divisão balanceada* [28]. A otimização de *Divisão Balanceada* define que a matriz poderá ser dividida tanto em sua linha central como em sua coluna central. A ideia desta otimização é que a maior dimensão de cada partição seja dividida a cada iteração, prevenindo que partições com lados muito desproporcionais sejam mantidas por muitas iterações. A Figura 3.13 apresenta um exemplo utilizando a divisão balanceada e não balanceada. Após a segunda iteração, a divisão não balanceada cria uma partição com tamanho maior que o tamanho máximo (max), exigindo uma iteração extra. Já na divisão balanceada, isso não ocorre.

Em um instante posterior, o *matching* por objetivo e a *execução ortogonal* foram incorporadas no estágio 4, gerando a versão CUDAlign-Stage4-Balanced [29]. Finalmente, foi gerada a última versão, CUDAlign-Stage4-Optimized, com o procedimento de *matching* otimizado (Figura 3.11), que reduz ainda mais a área processada (Figura 3.12) quando comparada com as versões anteriores (CUDAlign-Stage4-MM e CUDAlign-Stage4-Balanced).

A Figura 3.7d apresenta uma iteração do Estágio 4, resultando em novas coordenadas que formam partições ainda menores.

Estágio 5 O Estágio 5 alinha em CPU cada partição obtida no Estágio 4, utilizando o algoritmo Needleman-Wunsch (Seção 2.1). O alinhamento de cada partição é concatenado, resultando no alinhamento ótimo completo, conforme podemos ver na Figura 3.7e. Após a execução do Estágio 4, é garantido que os tamanhos das partições serão limitados ao



(a)

(b)

Figura 3.13: No algoritmo original de Myers-Miller (3.13a), a divisão é sempre feita na linha central, então 3 iterações foram necessárias para obter partições com tamanho menor que *"max"*. Com a divisão balanceada (3.13b), a divisão é feita na dimensão mais larga, então 2 iterações foram suficientes [7].

tamanho máximo de partição. Este tamanho deve ser pequeno suficiente para permitir que o Estágio 5 seja rápido. Visto que esse tamanho é constante, a complexidade do uso de memória em cada partição é constante e o alinhamento completo pode ser obtido com um consumo de memória linear.

Para reduzir o tamanho da saída do Estágio 5, um arquivo binário foi criado contendo informações sucintas sobre o alinhamento, que permitem representar um alinhamento sem armazenar as bases da sequência. Entretanto, para reconstruir o alinhamento é necessário que as sequências originais estejam disponíveis. O alinhamento completo pode ser visualizado por meio do Estágio 6.

Estágio 6 O Estágio 6 é utilizado opcionalmente para visualização da representação binária do alinhamento. Dadas as sequências de entrada S_0 e S_1, as coordenadas inicial (i_0 , j_0) e final (i_1 , j_1) e as listas de *gaps* GAP_1 e GAP_2, a reconstrução do alinhamento é feita por meio da união de todos os *gaps*. Iniciando com a coordenada (i, j) = (i 0 , j 0), o *gap* mais próximo é obtido das listas GAP_1 e GAP_2 e a próxima coordenada (i, j) será o final do *gap* mais próximo escolhido. Este procedimento é feito até que se encontre a posição final (i_1 , j_1). Esse procedimento permite que a representação textual (Figura 3.7f) seja obtida.

Os resultados obtidos pelo CUDAlign2.0 na GPU Nvidia GTX 285 comparando sequências com tamanho de até 33 MBP mostram um máximo de 23,63 GCUPS.

CUDAlign2.1

No CUDAlign 2.1 [6], a otimização *Block Pruning* (BP) é proposta para acelerar o processamento da matriz de programação dinâmica na obtenção do alinhamento local ótimo com uma GPU. Dependendo da similaridade entre as sequências, essa otimização permite que blocos de células da matriz de programação dinâmica sejam descartados. Com isso, elimina-se o cálculo de blocos de células que comprovadamente não contribuem para o alinhamento ótimo.

O *Block Prunning* leva em conta o maior escore já obtido no alinhamento e, antes de efetuar o cálculo da pontuação de uma célula verifica se tal operação é de fato necessária. Caso, a partir dessa célula, não se possa atingir um escore que seja ao menos igual ao escore já obtido, certamente essa célula não faz parte do alinhamento ótimo e seu valor é insignificante para o resultado final. O mesmo procedimento pode ser utilizado para um bloco de células ao invés de uma única célula, com a vantagem de que o procedimento aplicado a blocos é muito mais rápido, pois a verificação da condição de *pruning* é feita uma única vez para todo o bloco, considerando o pior caso onde a célula com maior escore encontra-se no canto superior direito do bloco. Entretanto, se o bloco for muito grande, o procedimento de *pruning* será pouco granular, dificultando o procedimento de *pruning*. No CUDAlign 2.1, o *Block Pruning* é aplicado no estágio 1, onde o escore ótimo ainda não é conhecido, acelerando o estágio mais longo do processamento.

Com a otimização do *Block Pruning*, o CUDAlign2.1 foi capaz de eliminar até 62.7% do cálculo das células da matriz de programação dinâmica, dependendo da similaridade entre as sequências. Os resultados coletados na GPU NVidia GTX 560 Ti comparando sequências de até 33 MBP mostram um máximo de 58,21 GCUPS.

CUDAlign3.0

O CUDAlign 3.0 [5] é capaz de executar o algoritmo de Gotoh (Seção 2.3) na comparação entre sequências muito longas utilizando diversas GPUs com paralelismo de granularidade fina e tem como saída o escore ótimo. Nessa versão, foi proposta uma nova arquitetura capaz de distribuir o processamento de uma única comparação em várias GPUs simultaneamente. Desta forma, o poder computacional das GPUs é somado com o intuito de acelerar o tempo de execução do estágio 1, que é a etapa mais demorada do CUDAlign.

Na arquitetura desenvolvida para o CUDAlign 3.0, cada GPU é responsável por calcular um intervalo de colunas da matriz de programação dinâmica. O tamanho de cada intervalo de colunas é escolhido de forma a igualar o número de linhas processadas por segundo em cada GPU, criando então uma distribuição balanceada de carga entre todas

as GPUs. Caso as GPUs possuam o mesmo poder de processamento, as colunas são distribuídas uniformemente, mas, em caso de GPUs heterogêneas, as colunas são distribuídas de acordo com a proporção entre suas capacidades de processamento.

As GPUs vizinhas transferem células de suas colunas divisórias. Caso essa transferência não seja bem projetada, o tempo de execução das demais GPUs será impactado. A Figura 3.14 ilustra uma divisão uniforme da matriz de programação dinâmica entre 4 GPUs. *Buffers* circulares são usados para sobrepor a computação e a comunicação, conectando os nós da GPU com os *sockets* TCP.

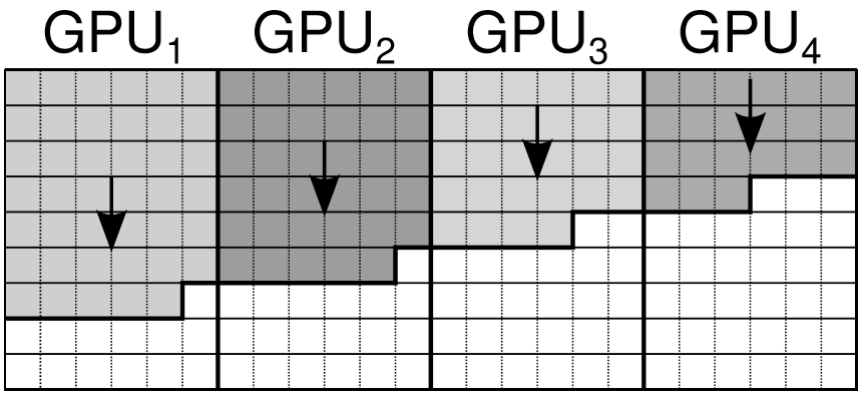


Figura 3.14: Divisão de colunas entre múltiplas GPUs [5].

Os resultados coletados no *cluster* Minotauro de GPUs com até 64 GPUs NVidia Tesla M2090 apresentam um máximo de 1726,47 GCUPS enquanto comparava-se sequências de DNA de 228 MBP × 249 MBP. Esta comparação levou 9 horas e 9 minutos para ser completada, com 64 GPUs.

CUDAlign4.0

O CUDAlign 4.0 [30] é capaz de recuperar alinhamentos longos com múltiplas GPUs. Para tanto, executa o CUDAlign 3.0 para obter o escore ótimo. A recuperação do alinhamento local ótimo é feita a partir da GPU que calculou o escore ótimo (GPU_x), comunicando-se com a GPU à esquerda (GPU_{x-1}) até que uma célula da matriz com valor zero seja encontrada.

Para a obtenção do alinhamento completo utilizando múltiplas GPUs, foram desenvolvidas duas estratégias de *traceback*. A primeira estratégia, utilizada como *baseline*, foi chamada de *Pipelined Traceback* (PT), onde cada GPU executa o *traceback* em um *pipeline* de 3 estágios (estágios 2 a 4 do CUDAlign), sendo que o estágio 2 é executado de maneira inerentemente serial. A segunda estratégia, chamada de *Incremental Speculative Traceback* (IST), utiliza as GPUs ociosas no estágio 2 para especular a localização do alinhamento ótimo, com o objetivo de acelerar sua execução. Sendo assim, ao terminar o

cálculo do estágio 2, a GPU_i informa à GPU_{i-1} a célula que deve ser utilizada como *crosspoint*: caso esse valor tenha sido corretamente especulado, o resultado é imediatamente informado à GPU_{i-2} e assim sucessivamente.

Resultados coletados no *cluster Keeneland Full Scale System* (KFS) com até 384 GPUs Tesla M2090 mostram um máximo de 10.370 GCUPS, ao se comparar as mesmas sequências do CUDAlign 3.0 (228 MBP × 249 MBP). O alinhamento ótimo foi obtido em menos de duas horas. A estratégia especulativa de recuperação do alinhamento, IST, chega a obter um *speedup* de 21,03x em relação à técnica *Pipelined Traceback*, reduzindo o tempo de execução de 3508 segundos para 167 segundos. Entretanto, mesmo com a estratégia IST, o tempo para a recuperação do alinhamento foi até 46% do tempo total de execução do CUDAlign [30], como no caso do alinhamento *chr13*, mostrando que essa etapa ainda necessita de otimizações.

3.3 Análise Comparativa

A Tabela 3.1 apresenta um quadro comparativo entre as diversas soluções abordadas nesse capítulo. Na coluna "Ferramenta" é informada a ferramenta de referência para os dados apresentados. Na coluna "Ano" é indicado o ano de publicação do artigo em questão. Como pode ser visto, na década 2000, havia muitas ferramentas propostas para CPU. Na década 2010, diversas ferramentas para GPU foram propostas, inclusive na versão híbrida CPU + GPU. A coluna "Saída" apresenta o retorno do processamento de cada solução, que em alguns casos pode ser apenas o escore e em outros o escore e o alinhamento completo. Uma das primeiras ferramentas capazes de retornar o alinhamento e não apenas o escore foi a desenvolvida por Rajko-Aluru (Seção 3.1.1).

Na coluna "Tam. Máximo" é exposto o maior comprimento de sequência suportado pela ferramenta. As ferramentas que aceitam as sequências com os maiores comprimentos são o CUDAlign e o SW#. As ferramentas que realizam a comparação de protéinas possuem os menores valores dessa coluna, já que a maior sequência de proteína possui aproximadamente 35000 caracteres. Na coluna "GCUPS" é apresentado o valor máximo de *Giga Cell Updates Per Second* (Bilhões de Células Atualizadas Por Segundo). A ferramenta mais rápida em termos de GCUPS é a ferramenta CUDAlign em sua quarta versão e a ferramenta mais lenta, a SWMMX, é também a mais antiga solução listada. Em geral, as soluções que utilizam GPUs são as que apresentam os maiores valores de GCUPS.

Por último, na coluna "Plataforma" é informada a plataforma escolhida para realizar o processamento. Algumas ferramentas utilizam a CPU e outras a GPU para realizar o processamento, podendo ter mais de uma unidade executando o processamento simulta-

Tabela 3.1: Sequências usadas nos testes

Ferramenta	Ano	Saída	Tam. Máximo	GCUPS	Plataforma
Rajko-Aluru(3.1.1)	2004	Alinhamento	1.100.000	0,24	60 X CPU
Z-Align(3.1.2)	2008	Alinhamento	3.282.708	1,40	64 X CPU
SWMMX(3.1.3)	2000	escore	567	0,20	1 X CPU (SIMD)
StripSW(3.1.4)	2007	escore	567	3,00	1 X CPU (SIMD)
CUDASW++1.0(3.2.2)	2009	escore	5.478	16,10	GPU
CUDASW++2.0(3.2.2)	2010	escore	5.478	29,70	GPU
CUDASW++3.0(3.2.2)	2013	escore	35.213	196,20	2 X GPU + 4 X CPU
SW#(3.2.3)	2013	Alinhamento	32.799.110	65,20	GPU
CUDAlign1.0(3.2.4)	2010	escore	32.799.110	20,30	GPU
CUDAlign2.0(3.2.4)	2011	Alinhamento	32.799.110	23,63	GPU
CUDAlign2.1(3.2.4)	2013	Alinhamento	32.799.110	58,21	GPU
CUDAlign3.0(3.2.4)	2014	escore	248.956.422	1726,47	64 X GPU
CUDAlign4.0(3.2.4)	2016	Alinhamento	248.956.422	10370,00	384 X GPU

neamente. A ferramenta CUDASW++3.0 é capaz de realizar um processamento híbrido, usando tanto a CPU como a GPU, já a ferramenta CUDAlign 4.0 é capaz de realizar o processamento em até 384 GPUs.

Dessa forma, tendo em perspectiva os dados apresentados na Tabela 3.1, é possível comparar diretamente as ferramentas em termos da saída, taxa de GCUPS e da plataforma que foi utilizada, pois essas informações podem ser comparadas em termos absolutos entre as ferramentas, ou seja, sem que o tipo de sequência que a ferramenta alinha possa impactar nos dados obtidos.

Analisando a Tabela 3.1 também pode-se verificar que a ferramenta CUDAlign apresenta desempenho muito bom e boa escalabilidade. Entretanto, como já observado na Seção 3.2.4, embora o estágio 1 do CUDAlign4.0 já esteja bastante otimizado, a recuperação do alinhamento ainda compromete o desempenho da ferramenta quando várias GPUs são utilizadas.

Capítulo 4

Projeto da Estratégia Multi-Bloco com Faixa Ajustável

Na presente dissertação foi proposta uma estratégia para a aceleração da obtenção do alinhamento de sequências biológicas com base nos algoritmos de Fickett (Seção 2.5) e Myers-Miller (Seção 2.4). Na Seção 4.1 é apresentada a visão geral do projeto. Na Seção 4.2 é descrito como foi realizada a determinação do valor da faixa de computação para cada bloco. Na Seção 4.3 são apresentados cenários de *speedups* da estratégia Fickett-CUDAlign em relação à execução do algoritmo original de Myers-Miller. Na Seção 4.4 é descrito o pseudo-código do Fickett-CUDAlign e na Seção 4.5 é feita uma comparação teórica do método Fickett-CUDAlign e o método original de Fickett.

4.1 Visão Geral

O objetivo da estratégia Fickett-CUDAlign desenvolvida nessa dissertação, é acelerar o estágio 4 do CUDAlign (Seção 3.2.4) ou de algoritmos que são baseados no Myers-Miller (Seção 2.4) onde se conheça previamente os escores das células no topo à esquerda e da base à direita em cada bloco.

Desse modo, a nossa estratégia de cálculo da matriz de programação dinâmica visa reduzir o tempo da comparação de sequências biológicas ao não realizar o cálculo do escore de regiões que estejam afastadas do alinhamento ótimo, com a garantia de que a sua eliminação não irá impedir que o alinhamento ótimo seja encontrado.

Para que isso seja possível, foi utilizado o algoritmo de Myers-Miller (Seção 2.4) combinado com o algoritmo de Fickett (Seção 2.5), possibilitando a recuperação do alinhameto em apenas uma dada faixa de tamanho D, assim como mostrado na Figura 4.1, ao invés do cálculo da matriz toda, como no algoritmo Myers-Miller original.

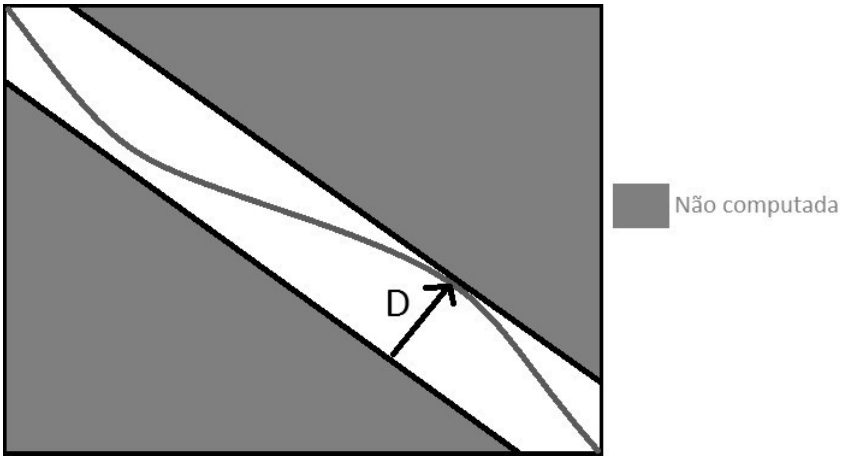


Figura 4.1: Algoritmo de Fickett com processamento do alinhamento na faixa de tamanho D.

Durante o estudo de comparações de várias sequências biológicas percebeu-se que a distância do alinhamento ótimo em relação à diagonal principal da matriz de programação dinâmica possui grande variação e a utilização do seu valor máximo em toda a comparação, conforme o proposto por Fickett, reduz o *speedup* da aplicação consideravelmente, como é possível observar na Figura 4.2.

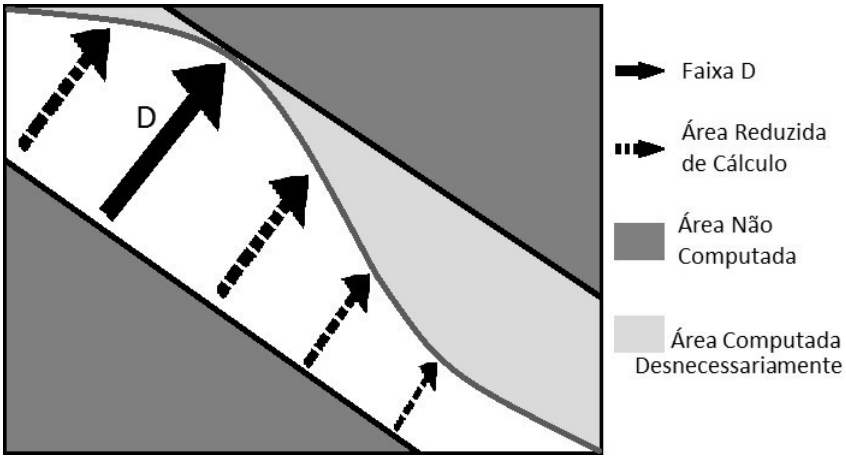
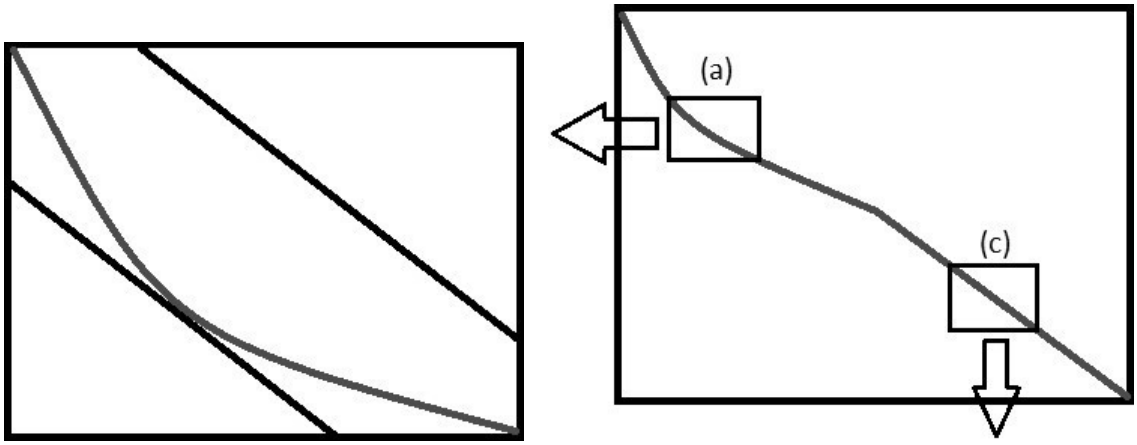


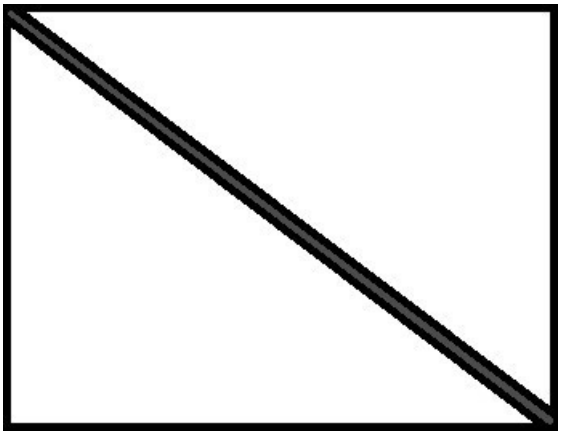
Figura 4.2: A faixa do algoritmo de Fickett pode ser muito superior ao necessário em grande parte do alinhamento.

Por esse motivo, passou-se a realizar o cálculo do valor da faixa de Fickett por blocos, ao invés de utilizar apenas um valor para toda a comparação. O resultado é a aplicação ajustável do tamanho da faixa em diferentes subsequências do alinhamento, como pode ser visto na Figura 4.3. Essa abordagem permite que a faixa seja reduzida em regiões onde o alinhamento possui poucos *gaps* (Figura 4.3c) e aumentada em regiões com mais *gaps* (Figura 4.3a), ajustando-se à forma do alinhamento.



(a) Faixa mais larga.

(b) Cálculo do alinhamento por blocos.



(c) Faixa mais estreita.

Figura 4.3: Algoritmo de Fickett com faixa de tamanho ajustável.

Além disso o algoritmo de Fickett foi pensado para ser utilizado com o algoritmo de Needleman-Wunsch, que é realizado em um único sentido da sequência (Seção 2.1). Já a estratégia desenvolvida na presente dissertação foi elaborada para o algoritmo de Myers-Miller, que possui complexidade de espaço $O(m + n)$ (Seção 2.4), possibilitando seu uso em sequências longas e permitindo que o alinhamento seja calculado paralelamente nos dois sentidos (tanto do começo para o fim como do fim para o começo da faixa de cada bloco). A Figura 4.4 mostra a diferença do processamento de uma subsequência usando a estratégia Fickett-CUDAlign em relação ao algoritmo de Fickett original na tabela de programação dinâmica, sendo que o algoritmo de Fickett processa o bloco em apenas um sentido, diferentemente do que pode ser visto na estratégia Fickett-CUDAlign.

Embora o algoritmo de Fickett esteja associado à utilização de uma faixa, nesse algoritmo não existem elementos que permitam definir o tamanho da faixa e que garantam que não seria necessário realizar todo o alinhamento novamente, como já foi explicado na Seção 2.5. Para que fosse garantido que o tamanho da faixa jamais deixaria de englobar

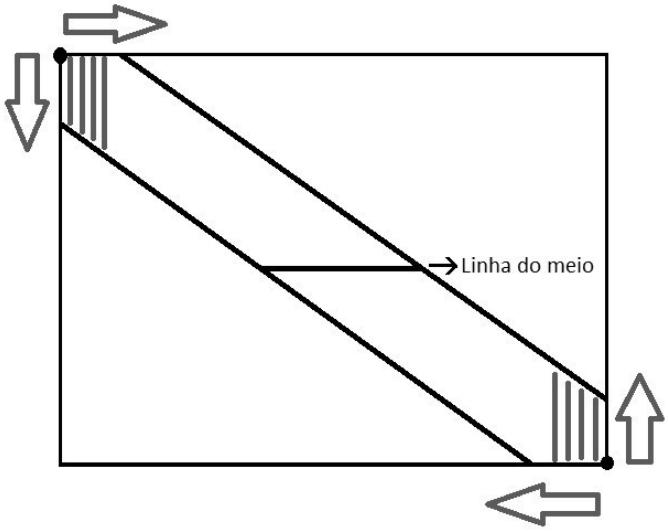


Figura 4.4: Método de Fickett processando um bloco com faixa de tamanho ajustável.

o alinhamento ótimo, foi desenvolvido um estudo na Seção 4.2 capaz de produzir uma equação que gere uma faixa que garantidamente contenha o alinhamento ótimo.

Assim, mesmo que no algoritmo original de Fickett não haja uma maneira precisa de se determinar o valor da faixa, usando a variação do algoritmo de Fickett proposta nesse capítulo tornar-se-há exequível o alinhamento de sequências com faixa ajustável garantindo que os limites da área a ser computada sempre irão respeitar o alinhamento e impedindo que seja necessário recalcular a faixa de processamento.

4.2 Determinação do Tamanho da Faixa

O algoritmo de Fickett calcula a matriz de programação dinâmica para uma única faixa de anti-diagonais (Seção 2.5). A determinação de um valor apropriado para essa faixa é de grande importância para o desempenho do método. No Fickett-CUDAlign, admite-se que os escores do canto superior à esquerda e do canto inferior à direita são conhecidos ($score_i$ e $score_f$) e nós utilizaremos essa informação para a determinação de tamanhos ajustáveis para a faixa.

Para cada bloco, a determinação do valor da faixa a ser utilizada pelo método de Fickett-CUDAlign é feita por uma equação que usa quatro termos: $perfect\_match$, $score_{diff}$, $desvio\_perfect\_match$ e min_gaps.

No termo *perfect_match* é calculado o escore máximo que o bloco poderia obter caso houvesse um *perfect match* (todos os caracteres iguais) entre as subsequências. Nesse cálculo, temos que considerar o caso onde o tamanho das subsequências não é o mesmo, ou seja, os blocos são retangulares, e não quadrados. Sendo assim, o tamanho da menor

sequência é multiplicado pela pontuação do *match* (*perfect match* na menor subsequência) decrescido da diferença entre o tamanho das duas subsequências multiplicado pelo módulo da pontuação da extensão do *gap*. Assim, calcula-se o caso do alinhamento com *perfect matches*, que é descrito na Equação (4.1). A Figura 4.5 exemplifica esse caso.

$$perfect\_match = min(m_b, n_b) * match - (max(m_b, n_b) - min(m_b, n_b)) * |gap\_ext| \quad (4.1)$$

Onde m_b e n_b são os tamanhos dos trechos das subsequências utilizadas no bloco, gap_ext é o valor da extensão do *gap* e que *match* é a pontuação do *match*.

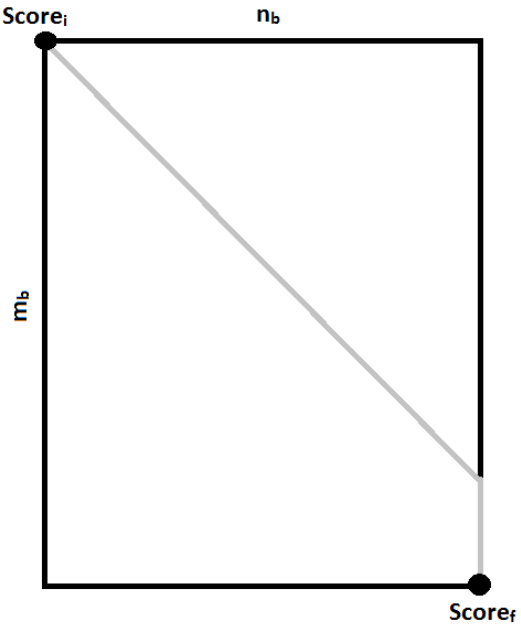


Figura 4.5: Exemplo de ocorrência de *perfect match* em sequências de tamanho diferente. O número de *gaps* é exatamente o igual à diferença de tamanho entre as subsequências (m_b - n_b).

No segundo termo ($score\_diff$) o valor obtido é a diferença de escore entre o ponto inicial e o final do bloco, para que se possa avaliar o quão distante do caso ótimo o escore se encontra (Equação (4.2)).

$$score\_diff = score_f - score_i \quad (4.2)$$

Onde $score_f$ e $score_i$ são os valores dos escores do último e do primeiro ponto do alinhamento das subsequências que compõem o bloco.

No terceiro termo ($desvio\_perfect\_match$) o resultado é dividido por $2 * |gap\_ext| + match$, como pode ser visto na Equação 4.3, pois a cada desvio do alinhamento ótimo a diferença mínima na pontuação é de $2 * |gap\_ext| + match$ pontos, que corresponde às duas extensões de *gap* necessárias para realizar um desvio e retornar ao alinhamento das duas sequências e uma pontuação de *match* que é perdida ao efetuar o desvio (Figura 4.6).

$$desvio\_perfect\_match = 2 * |gap\_ext| + match \tag{4.3}$$

Onde gap_ext é o módulo da penalidade de *gap*.

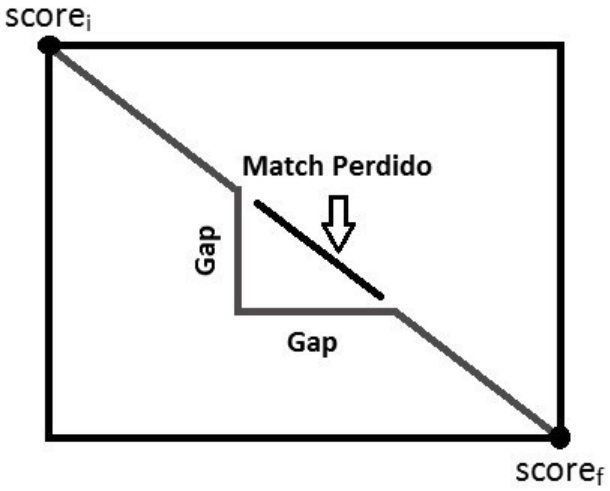


Figura 4.6: A cada desvio do alinhamento ótimo a diferença mínima na pontuação é de $2 * gap\_ext + match$ pontos.

Finalmente no quarto termo (min_gaps), é calculada a diferença entre o tamanho das subsequências, pois ela indica o número mínimo de *gaps* para que a faixa englobe o alinhamento ótimo entre as sequências, como descrito na Equação(4.4).

$$min\_gaps = max(m_b, n_b) - min(m_b, n_b) \tag{4.4}$$

A equação para cálculo da faixa é então ilustrada na Equação 4.5:

$$Band = \left\lceil \frac{perfect\_match - score\_diff}{desvio\_perfect\_match} \right\rceil + min\_gaps \tag{4.5}$$

Na Equação 4.5 o valor do $perfect\_match$ é subtraído pelo valor do $score\_diff$ para encontrar a diferença entre a pontuação máxima que a sequência poderia obter e a pontuação efetivamente obtida. Esse valor é dividido pelo $desvio\_perfect\_match$ para que

se possa conhecer quantos *gaps* a mais que o necessário podem ser encontrados na comparação. Ao final, o valor resultante dessa divisão, arredondado para cima, é adicionado de min_gaps para encontrar a quantidade total de *gaps* que podem ser localizados na comparação realizada e, dessa forma, descobrir o alcance que a faixa de Fickett deve ter para englobar todo o alinhamento ótimo.

Para elaborar a equação (4.5), levamos em conta o fato de que o alinhamento passa necessariamente pelo primeiro e pelo último ponto de cada bloco ($score_i$ e $score_f$). Além disso, assumimos que o escore do alinhamento nesses pontos já esta disponível nessa fase da execução.

A equação (4.5) ainda pode ser re-escrita expandindo-se os termos $perfect\_match$ e min_gaps e simplificando as operações conforme a Equação (4.6), dando origem à Equação (4.7).

$$Band = \left\lceil \frac{(max(m_b, n_b) * match + max(m_b, n_b) * |gap\_ext| - min(m_b, n_b) * |gap\_ext| - score\_diff)}{desvio\_perfect\_match} \right\rceil \quad (4.6)$$

$$Band = \left\lceil \frac{(max(m_b, n_b) * match + (min\_gaps * |gap\_ext|) - score\_diff)}{desvio\_perfect\_match} \right\rceil \quad (4.7)$$

Na confecção das equações levamos em conta o pior caso. Logo, cabe ressaltar que valores menores para a faixa não garantem que o alinhamento ótimo seja encontrado.

Para aplicar o valor da faixa no alinhamento outro fato também teve de ser observado: Como o algoritmo de Myers-Miller encerra o processamento de cada parte na linha do meio, algumas das células que seriam candidatas ao processamento pela estratégia proposta podem ser eliminadas dessa fase por excederem os limites inferior ou superior do algoritmo de Myers-Miller. A Figura 4.7 fornece um exemplo de eliminação do processamento de células ao encontrar o escore-alvo, semelhantemente ao proposto no CUDAlign (Seção 3.2.4).

Essa modificação faz com que o procedimento de *matching* ortogonal seja alterado, pois diferentemente do proposto no CUDAlign os dois lados a serem processados são limitados ao valor da faixa proposta na Equação (4.5).

Na Figura 4.9 é possível analisar o momento exato no qual o escore-alvo é encontrado. Comparando com a Figura 3.11 é possível notar que além de não processar as células posteriores ao escore-alvo o Fickett-CUDAlign reduz também a área a ser processada antes de encontrar o escore-alvo.

É importante destacar que o resultado obtido para o comprimento da faixa pode não ser o valor mínimo de faixa necessário para realizar o alinhamento, pois o valor exato de *gaps* em cada direção é desconhecido, assim como a quantidade de *mismatches* presentes

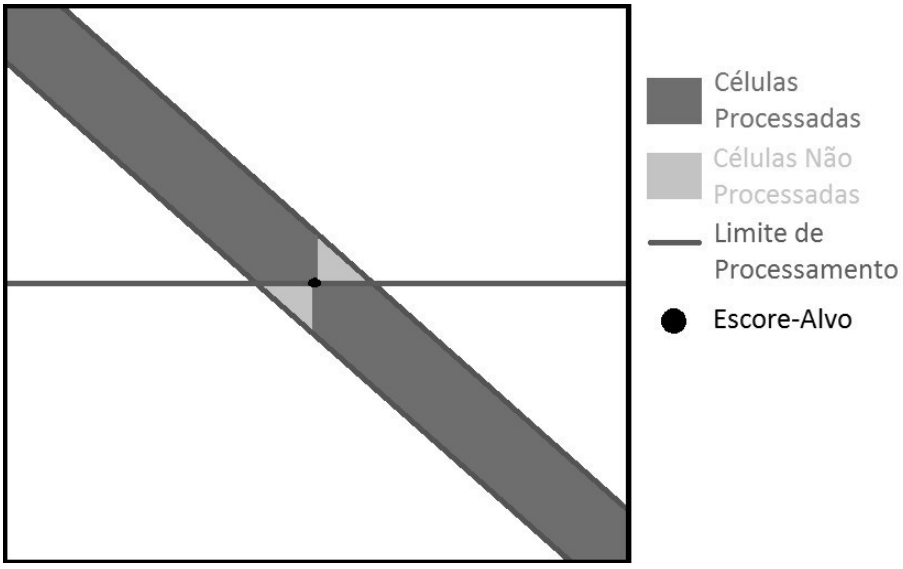


Figura 4.7: Exemplo de eliminação de células que excedem os limites do algoritmo Myers-Miller.

no bloco a ser calculado. Isso leva a uma variação nos valores que podem ser produzidos, pois não é possível saber a direção do *gap* previamente (acima ou abaixo do alinhamento ótimo). Na Equação 4.7 os valores são considerados para uma única direção e depois a faixa é computada com esse valor para ambas as direções (da parte superior esquerda para a superior direita e da parte inferior direita para a parte inferior esquerda). Logo se houver *gaps* em direções diferentes, o valor da direção onde houver a menor quantidade de *gaps* será um excedente.

Dessa forma, nota-se que existem três diferentes tipos de células a serem consideradas ao utilizar a abordagem Fickett-CUDAlign:

Celulas Candidatas: As células candidatas são aquelas que, após realizado o cálculo do valor da faixa, encontram-se dentro dos limites superior e inferior determinados pela faixa de computação a ser utilizada no cálculo de comparação das sequências. Essas células são candidatas a serem processadas pela estratégia Fickett-CUDAlign.

Células Processadas: São as células que de fato serão processadas durante a execução da estratégia Fickett-CUDAlign. Todas as células processadas são células que foram marcadas como candidatas após a fase de determinação do tamanho da faixa e não foram eliminadas do processamento após se encontrar o escore-alvo.

Células Não-Processadas: São as células que foram eliminadas após o cálculo de determinação do tamanho da faixa ou são as células candidatas que foram eliminadas do

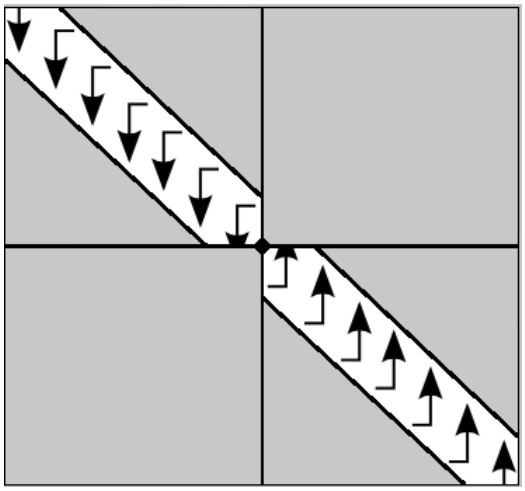


Figura 4.8: Execução ortogonal Fickett-CUDAlign. A área em cinza não precisa ser processada, pois a faixa de computação determinada pela equação (4.5) e a utilização do método ortogonal descartaram essa área da fase de processamento.

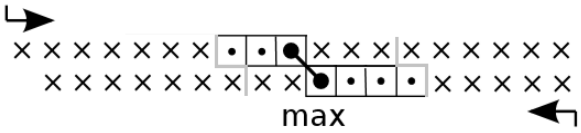


Figura 4.9: Detalhe no procedimento de *matching* ortogonal utilizando as faixas de computação determinadas pela equação (4.5).

processamento após o escore-alvo ter sido encontrado.

Na Figura 4.10 tem-se um exemplo da utilização da equação de determinação do valor da faixa para um bloco a ser computado. Nessa figura o *gap* já foi aberto no bloco anterior e um *mismatch* ocorre no alinhamento. Considerando o valor da penalidade de extensão *gap* como -2, a penalidade de *mismatch* como -3, o valor do *match* como +1 e tendo a Equação 4.5 como base, foi determinado o valor da faixa a ser utilizado pela estratégia Fickett-CUDAlign. Inicialmente, é determinado o valor do $perfect\_match$ entre as sequências, que calcula a pontuação da maior diagonal e reduz com a a penalidade de *gap* a diferença de comprimento entre as sequências, como pode ser visto na Equação (4.8).

$$perfect\_match = 9 * 1 - (10 - 9) * |-2| = 7 \tag{4.8}$$

O valor dos escores obtidos nas duas pontas do bloco são subtraídos conforme a Equação 4.9:

$$score\_diff = 70 - 67 = 3 \tag{4.9}$$

A seguir, usam-se as pontuações de *gap_ext* (-2) e *match* (+1) para se calcular o desvio do *perfect match* (Equação 4.10), revelando que o escore da sequência diferiu em 4 do *perfect_ match*. Considerando o impacto de um desvio no alinhamento, conforme visto na Figura 4.6 é descoberto o valor do desvio do *perfect_ match*, que leva em consideração as pontuações que estão sendo adotadas (Equação (4.10)). O resultado da divisão entre o valor da diferença do *perfect_ match* e o valor de desvio do *perfect_ match* é $0,8$, que será interpretado como um *gap* no alinhamento das sequências.

$$desvio\_perfect\_match = 2 * |-2| + 1 = 5 \tag{4.10}$$

Depois, analisado o tamanho das sequências, como o tamanho das sequências difere em um elemento é necessário que seja feito um ajuste para encontrar a diagonal exata do caso *perfect_ match*, aumentando o tamanho da faixa, conforme mostra a Equação (4.11). Assim os valores obtidos de mínimo de *gaps* e de *gaps* encontrados pela estratégia no alinhamento são somados na Equação (4.12), resultando em um valor de faixa igual a 2, a ser aplicado nos dois lados do *perfect_ match*.

Calcula-se então o quarto termo (min_gaps) com a Equação 4.11.

$$min\_gaps = max(10, 9) - min(10, 9) = 1 \tag{4.11}$$

Finalmente, o tamanho da faixa é calculado com a Equação 4.12.

$$Band = \left\lceil \frac{(7-3)}{5} \right\rceil + 1 = 2 \tag{4.12}$$

Como já discutido nessa seção, embora o *gap* tenha ocorrido em apenas uma direção, a estratégia calcula o mesmo valor da faixa para os dois lados do alinhamento por falta de indicação da orientação do *gap*.

Na Figura 4.11 estão destacadas em preto as células que foram consideradas candidatas a serem processadas pela estratégia Fickett-CUDAlign após a determinação da faixa a ser computada. Essas células se apresentam dentro do limite fixado na Equação (4.12).

Na Figura 4.12 estão destacadas em preto as células que foram processadas pela estratégia Fickett-CUDAlign para encontrar o escore-alvo. Algumas células que foram marcadas como candidatas ao processamento não foram processadas; essas células estão destacadas de cinza.

Como o algoritmo de Myers-Miller é aplicado recursivamente sobre a comparação das sequências para que o alinhamento ótimo seja encontrado, as faixas utilizadas na estra-

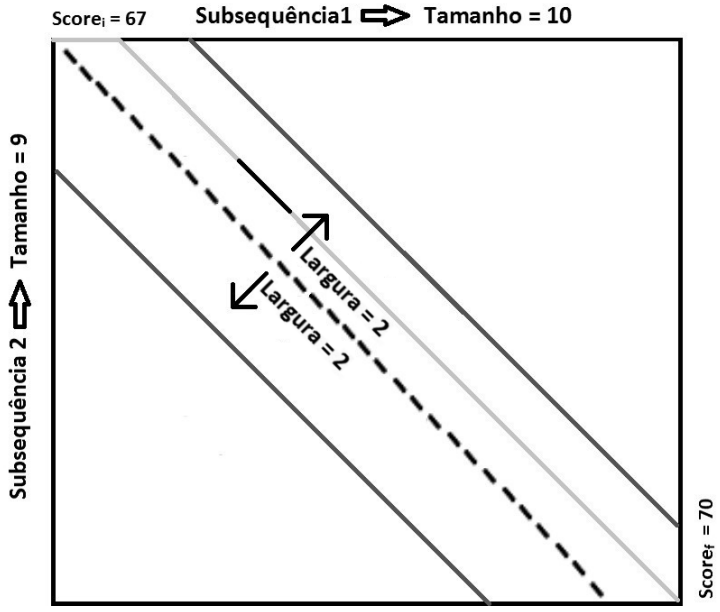


Figura 4.10: Exemplo de aplicação da Equação (4.5) para a determinação do valor da faixa.

tégia Fickett-CUDAlign também são aplicadas recursivamente na matriz de programação dinâmica e a cada iteração o número de blocos aumenta exponencialmente $O(2^n)$ e o valor da faixa se torna mais próximo do alinhamento a ser obtido. A Figura 4.13 apresenta como o valor do tamanho da faixa é alterado em uma, duas e três recursões da estratégia Fickett-CUDAlign.

Como o valor da diferença dos escores irá compor a equação do comprimento da faixa, o número de *mismatches* impacta diretamente o desempenho da estratégia, pois sequências com maiores quantidades de *mismatches* apresentarão um falso positivo para *gaps*, pois não é possível distingui-los por falta de informação. A Figura 4.14 reflete essa situação.

Da mesma forma que não é possível identificar a direção dos *gaps* e nem identificar os *mismatches*, também não é possível inferir a presença do gap_{open}. Por isso, o gap_{open} e o *mismatch* não fazem parte da Equação (4.7).

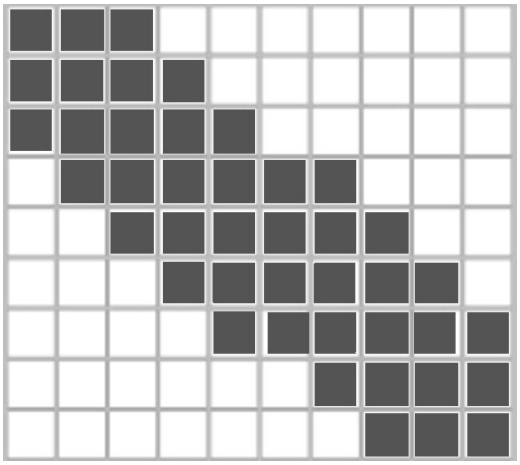


Figura 4.11: Células candidatas (em preto) a serem processadas pela estratégia Fickett-CUDAlign.

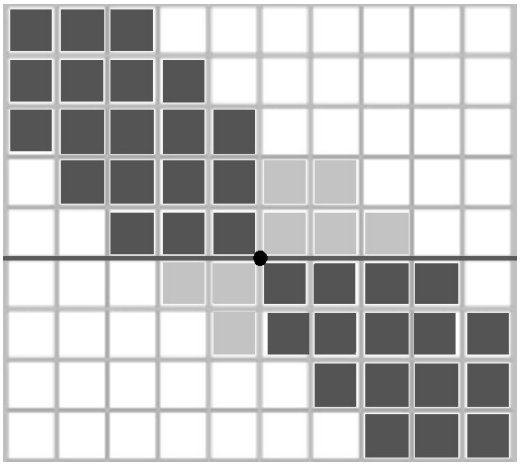


Figura 4.12: Células que de fato foram processadas (em preto) durante a execução da estratégia Fickett-CUDAlign.

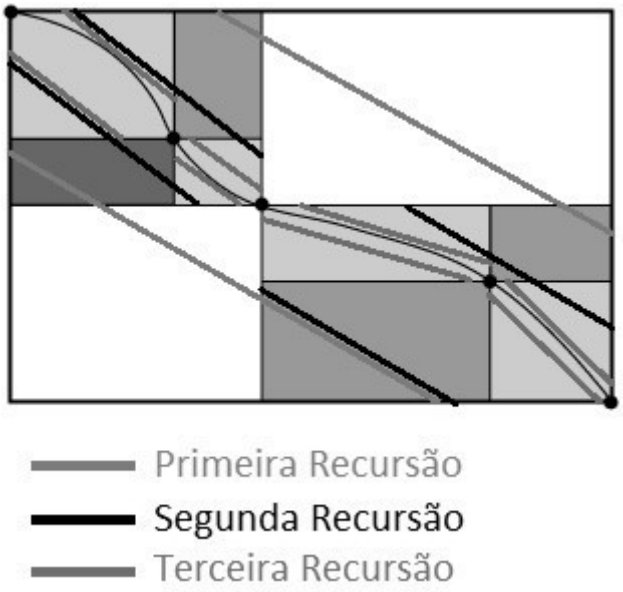


Figura 4.13: Variação do tamanho da faixa de acordo com a aplicação da Equação (4.5) recursivamente.

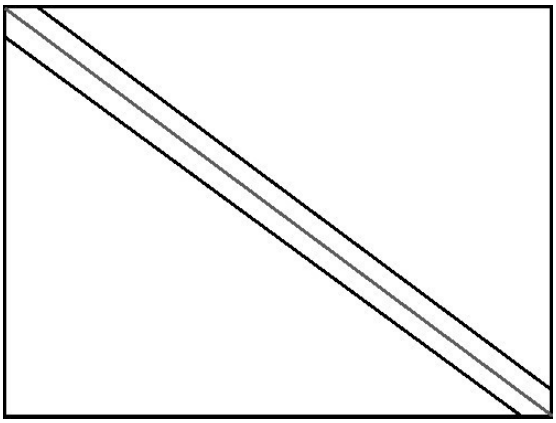


Figura 4.14: Os *mismatches* seguem a mesma direção do *perfect match*, mas alargam a faixa de cálculo do alinhamento.

4.3 Ganho de Desempenho

O ganho de desempenho da estratégia é muito dependente do grau de similaridade das subsequências, visto que a determinação do valor da faixa de computação é feita utilizando exclusivamente esse fator. Primeiramente é considerado o grau de similaridade das subsequências comparadas baseado no escore apresentado e depois é aferido o grau de similaridade levando em conta a diferença no comprimento das subsequências. Todavia, considerando os valores utilizados para as pontuações de *match*, *mismatch*, $gap_{extension}$ e gap_{open} é possível calcular aproximadamente o *speedup* médio do método Fickett-CUDAlign sobre o algoritmo de Myers-Miller.

Assumindo o valor da pontuação do *match* igual a +1, o valor da pontuação do *mismatch* igual a -3, o valor da abertura de *gaps* -3 e o valor da pontuação do *gap_extension* igual a -2, quando um desvio ocasionado por um *gap* é encontrado na sequência, necessariamente existem quatro *matches* para compensá-lo, pois o alinhamento local não assume valores negativos, gerando um *speedup* de aproximadamente 2.5 vezes. Já o *mismatch* causa um falso positivo de 0.6 *gap*, mas afeta em apenas 3 a pontuação, necessitando de mais 3 *matches* para compensá-lo, assim gerando um *speedup* de aproximadamente 3.33 vezes. Considerando um gap_{open}, o mesmo seria tratado como um falso positivo de 1.5 *gap* e necessitaria retornar ao alinhamento principal com o mesmo valor de falso positivo. Como é necessário pelo menos três *matches* para compensá-lo, ele gera um *speedup* de aproximadamente 3.0 vezes. O caso trivial, que é o *perfect match*, é resolvido em tempo linear $O(n)$, considerando n o tamanho da maior sequência, pois no caso trivial o tamanho da faixa é 1 e o número de células da matriz de programação dinâmica que serão computadas é exatamente o tamanho da maior sequência. Por isso, em um alinhamento com tamanho e quantidade de blocos suficientemente grande, o tempo de execução no *perfect match* tem complexidade $O(n)$ ao invés de $O(mn)$.

Com isso, podemos afirmar que o valor do *speedup* do Fickett-CUDAlign em um alinhamento local com tamanho e quantidade de blocos suficientemente grande, a serem discutidos no Capítulo 5, seja de aproximadamente $2,5$ a $3,33$ vezes em relação ao algoritmo de Myers-Miller original no caso médio. No melhor caso o *speedup* é proporcional ao tamanho das sequências comparadas e, no pior caso, o número e o tamanho de blocos é muito pequeno. Por isso, a faixa não é capaz de reduzir a área de computação de maneira eficiente e a estratégia Fickett-CUDAlign será executada praticamente da mesma forma que o algoritmo original de Myers-Miller.

Em relação ao algoritmo de original de Fickett, não é possível realizar uma comparação de ganho de desempenho, pois o algoritmo de Fickett não apresenta uma forma de calcular o *speedup* para o caso médio de execução do algoritmo, já que não é fixado como o tamanho da faixa é calculado.

4.4 Pseudo-Código

Para facilitar a compreensão do funcionamento do método Fickett-CUDAlign foi feito um pseudo-código (Algoritmo 1), que apresenta as entradas, saídas e procedimentos realizados para a obtenção dos pontos do alinhamento ótimo.

Algorithm 1 Algoritmo Fickett-CUDAlign

Entrada: Subsequências $S1$ e $S2$, Score $score_i$ e $score_f$
Saída: $crosspoint$

```
1: /*Calculo do tamanho da faixa*/
2: score_diff ← score_f − score_i
3: k ← calculate_band(S1, S2, score_diff)
4: seq1_length ← size(S1)
5: j ← 0
6: loop
7:    /*Calculo das extremidades da faixa*/
8:    upper_left ← Calculate_VBFickett_upper_left(j, k)
9:    lower_left ← Calculate_VBFickett_lower_left(j, k)
10:   upper_right ← Calculate_VBFickett_upper_right(seq1_length − j, k)
11:   lower_right ← Calculate_VBFickett_lower_right(seq1_length − j, k)
12:   /*Calculo das coordenadas do maior escore dentro da faixa de computacao*/
13:   crosspoint1[j] ← coordinates(MAX_(upper_left, lower_left, S1, S2))
14:   crosspoint2[j] ← coordinates(MAX_(upper_right, lower_right, S1, S2))
15:   if j > seq1_length/2 then
16:      crosspoint ← crosspoint1[j] + crosspoint2[j − Seq1_length − j]
17:      if check(crosspoint, score_diff) = TRUE then
18:         return crosspoint
19:      end if
20:      crosspoint ← crosspoint2[j] + crosspoint1[j − Seq1_length − j]
21:      if check(crosspoint, score_diff) = TRUE then
22:         return crosspoint
23:      end if
24:   end if
25:   j++
26: end loop
```

O Algoritmo 1 recebe como entrada duas subsequências S1 e S2, o $score_i$ que é o escore do ponto inicial da comparação e o $score_f$ que é o escore do ponto final da comparação. O processamento do algoritmo começa calculando o $score_{diff}$ que é a subtração do $score_f$ pelo $score_i$, ou seja, é a diferença entre os escores da última célula da última coluna e o da

primeira célula da primeira coluna (linha 2). Após isso, é determinando o valor que será utilizado para a faixa do método de Fickett, a função *calculate_band(S1, S2, score$_{diff}$)* calcula o valor da faixa a ser utilizada a partir do tamanho das duas subsequências e do valor do *score$_{diff}$*, aplicando a Equação (4.5). Supõe-se nesse algoritmo que o valor das pontuações de *match*, *mismatch* e *gap* sejam constantes.

Depois, é iniciado um *loop* (linha 6) que irá se repetir até que seja encontrado o *crosspoint* que pertence ao alinhamento ótimo. Internamente ao *loop*, baseado no valor da faixa são obtidas as fronteiras do cálculo do alinhamento usando as funções de fronteira *Calculate_VBFickett_upper_left* (linha 8) e *Calculate_VBFickett_lower_left* (linha 9) do lado esquerdo para o lado direito e *Calculate_VBFickett_upper_right* (linha 10) e *Calculate_VBFickett_lower_right* (linha 11) do lado direito para o lado esquerdo. O valor da fronteira é calculado utilizando tanto o valor j, que indica o posicionamento da coluna em relação à sequência, quanto o valor k, que determina o tamanho da faixa. Essas fronteiras servem para a eliminação de células candidatas que se encontram fora da faixa de computação do algoritmo de Myers-Miller e evitam que valores posteriores ao escore-alvo sejam processados.

Obtidos os valores das extremidades da faixa, são colocados nas listas de possíveis *crosspoints* os pontos que obtiveram os maiores valores em cada coluna calculada (linhas 13 e 14). Obedecendo ao algoritmo de Myers-Miller os valores máximos de escore gerados em cada linha são somados a partir do momento que chegam ao meio da sequência e, caso a soma corresponda ao valor do escore-alvo, as coordenadas daquele ponto são retornadas para serem gravadas no registo de *crosspoints* (linhas 18 e 22). As informações gravadas nesses *crosspoints* são as informações que serão utilizadas nas próximas recursões do algoritmo, pois cada *crosspoint* encontrado será utilizado como fim de um bloco e início do outro na próxima recursão, de acordo com o algoritmo de Myers-Miller (Seção 2.4).

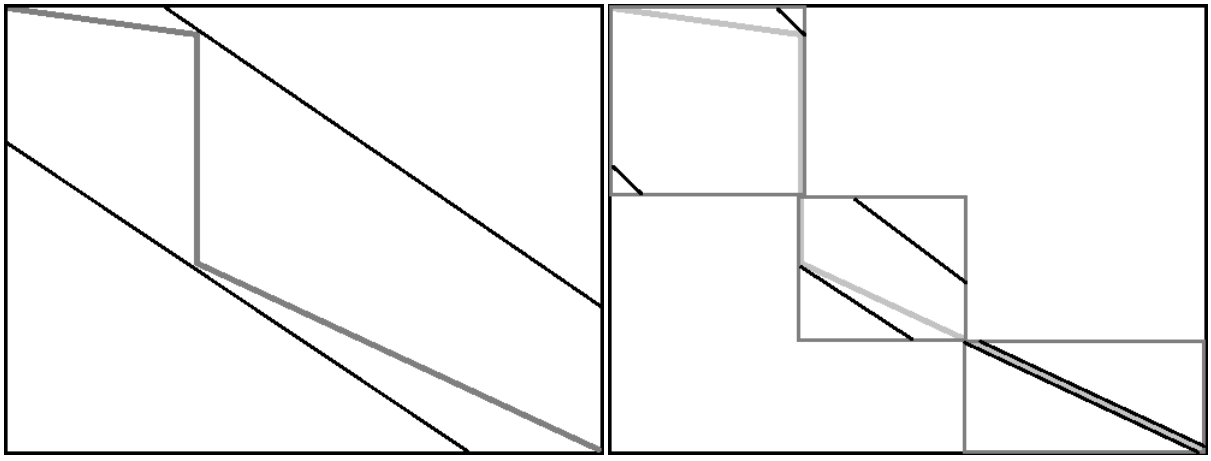
Utilizando o algoritmo 1 recursivamente é possível obter todos os pontos do alinhamento entre duas sequências biológicas, a exemplo do que é feito no algoritmo de Myers-Miller [22].

4.5 Comparação do Fickett-CUDAlign com o Fickett original

A estratégia Fickett-CUDAlign apresentada na Seção 2.5 possui a seguinte vantagem em relação ao método de Fickett original. O método original se limita a escolher um valor de faixa sem um critério bem definido e, posteriormente, tentar realizar o alinhamento; caso não seja possível, é necessário aumentar o valor da faixa, novamente sem um parâmetro de escolha e ir tentando sucessivamente até conseguir (Seção 2.5).

Por isso, não é possível determinar um *speedup* para o método de Fickett originalmente proposto, nem tampouco afirmar que tal *speedup* ocorrerá, haja vista a possibilidade de realizar várias vezes o alinhamento, de forma que o tempo utilizado para realizar o alinhamento seja maior que o tempo necessário sem o auxílio do método.

Mesmo conseguindo obter uma faixa que compreenda o alinhamento, o tamanho da faixa é fixo, e por isso regiões com muitos *gaps* podem comprometer severamente a utilização do método. Dessa forma, o método de Fickett se mostraria útil apenas em sequências que possuam uma quantidade de *gaps* muito pequena em relação ao tamanho da sequência. Já o método Fickett-CUDAlign, além de possuir um *speedup* teórico que determine um desempenho melhor do que a utilização do algoritmo original de Myers-Miller, é ajustável ao formato que o alinhamento ótimo possui, sem carregar influências do tamanho máximo da faixa de uma parte do alinhamento no tamanho da faixa de outra parte, já que processa o alinhamento em vários blocos ao invés de considerar todo o alinhamento para determinar o valor de uma única faixa. A Figura 4.15 mostra a diferença em termos adequação ao formato do alinhamento. A Figura 4.16 ressalta a diferença entre a área processada nas duas sequências.

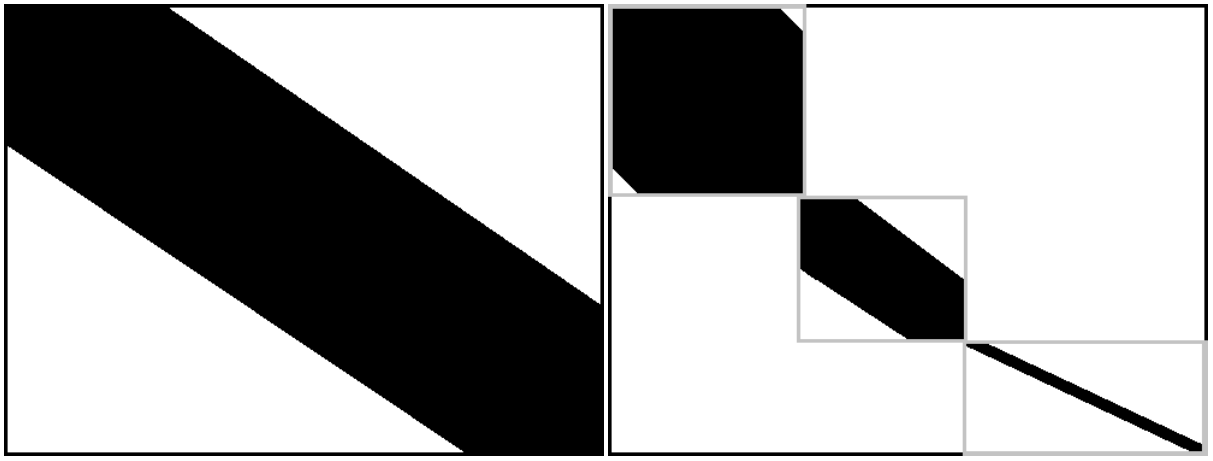


(a) Faixa gerada pelo método de Fickett original.

(b) Faixas geradas pelo método Fickett-CUDAlign.

Figura 4.15: Diferença entre o método Fickett-CUDAlign e o método de Fickett original na adequação ao formato do alinhamento ótimo.

Outro fato muito relevante a ser ressaltado a respeito do método de Fickett original é a respeito dos alinhamentos de grandes sequências, pois como não existe nenhuma forma de determinar o tamanho da faixa previamente, a utilização de um algoritmo recursivo como o de Myers-Miller se torna inviável. Isso acontece porque um valor escolhido para uma recursão é completamente inadequado para a próxima e para a anterior na maior parte dos casos, dada a grande diferença entre o tamanho dos blocos a serem processados



(a) Área processada pelo método de Fickett original.

(b) Área processada pelo método Fickett-CUDAlign.

Figura 4.16: Diferença entre o método Fickett-CUDAlign e o método de Fickett original na área de processamento.

em cada recursão. Entretanto, como a ideia que concebeu o algoritmo de Fickett não previa tal utilização, essa possibilidade sequer foi considerada.

Assim, acreditamos que o Fickett-CUDAlign seja muito mais rápido do que a proposta original de Fickett, pois é capaz de garantir o ganho de desempenho em relação ao algoritmo original de Myers-Miller por causa da equação do tamanho das faixas que calcula. Além disso, ele permite que sequências maiores tirem proveito de suas faixas variáveis e ajustáveis, pois foi desenvolvido para algoritmos que possuem complexidade espacial linear, visto que opera em blocos de maneira recursiva, e não no alinhamento inteiro, reduzindo a demanda por memória para computar o alinhamento.

Portanto, considerando as informações levantadas no presente capítulo, depreende-se que é importante não apenas ter um método que permita a redução da área de computação, mas também deve ser viável calcular como essa redução ocorrerá, para que haja garantia de que a obtenção do alinhamento ótimo não será prejudicada em hipótese alguma. Outrossim, é importante que a solução de redução da área de computação seja projetada para a utilização de algoritmos que possuem complexidade espacial $O(n)$ (linear), de maneira que a computação da comparação de sequências longas seja factível.

Capítulo 5

Resultados

O presente capítulo apresenta os resultados da estratégia Fickett-CUDAlign descrita no Capítulo 4. Na Seção 5.1, é descrito o ambiente computacional utilizado nos testes e, na Seção 5.2, são apresentadas as sequências utilizadas. Na Seção 5.3, são apresentados resultados de tempo de execução e *speedups* obtidos nos testes. Na Seção 5.4, é feito o estudo do *speedup* do Fickett-CUDAlign utilizando distintas quantidades de *threads* na computação. Na Seção 5.5, o comportamento dos alinhamentos obtidos é analisado e na Seção 5.6, é apresentada uma comparação entre os tempos de execução da ferramenta Fickett-CUDAlign e da ferramenta SW#.

5.1 Ambiente Computacional

A estratégia Fickett-CUDAlign foi implementada em C/C++ e um computador do laboratório LAICO (Laboratório de Sistemas Integrados e Concorrentes) da UnB foi usado em nossos testes. O computador é composto de um processador de 4 núcleos Intel Core i7 3770 com 8GB de memória RAM, 1TB de espaço de armazenamento e uma GPU Nvidia GTX680, com 1536 núcleos e 2GB de memória RAM.

Os parâmetros usados no cálculo do alinhamento foram: *ma* (*match*): +1; *mi* (*mismatch*): -3; *Gap Open* (G_{open}): -3; *Gap Extension* (G_{ext}): -2. Nas execuções do CUDAlign original, do Fickett-CUDAlign e do SW# utilizamos os seguintes parâmetros: Número de *threads* da CPU: 8; e tamanho final do bloco no estágio 4: 24x24.

As etapas 1 a 3 da execução da ferramenta CUDAlign (essas etapas foram explicadas na Seção 3.2.4) foram executadas na GPU, já as etapas 4 a 6 na CPU. Na etapa 4, especificamente, foi aplicada a estratégia Fickett-CUDAlign.

Tabela 5.1: Sequências usadas nos testes

Comparação	Sequência 1		Sequência 2		Escore Ótimo	
	Accession	Tam.	Accession	Tam.	Local	Global
10K x 10K	$AF133821.1$	10k	$AY352275.1$	10k	5091	2981
57K x 57K	$NC\_001715.1$	57k	$AF494279.1$	57k	52	-63880
162K x 172K	$NC\_000898.1$	162k	$NC\_007605.1$	172k	18	-208667
543k x 536K	$NC\_003064.2$	543k	$NC\_000914.1$	536k	48	-585725
1M x 1M	$CP000051.1$	1M	$AE002160.2$	1M	88353	-1189459
3M x 3M	$BA000035.2$	3M	$BX927147.1$	3M	4226	-2662376
5M x 5M	$AE016879.1$	5M	$AE017225.1$	5M	5220960	5220950
7M x 5M	$NC\_005027.1$	7M	$NC\_003997.3$	5M	172	-8201748
10M x 10M	$NC\_017186.1$	10M	$NC\_014318.1$	10M	10235188	10235188
23M x 25M	$NT\_033779.4$	23M	$NT\_037436.3$	25M	9063	-27446770
47M x 32M	$NC\_000021.7$	47M	$BA000046.3$	32M	27206434	-572719

5.2 Sequências utilizadas nos testes

Nos nossos testes, foram comparadas sequências reais de DNA obtidas do *National Center for Biotecnology Information (NCBI)* no *site www.ncbi.nih.gov*. O tamanho das sequências variou de 10KBP (Milhares de Pares de Base) até 47MBP (Milhões de Pares de Base). Para efeitos de validação, os escores ótimos obtidos durante os nossos testes também são mostrados.

A Tabela 5.1 apresenta informações relevantes sobre as sequências que foram utilizadas nos testes, dentre elas o tamanho de cada sequência, o código de acesso dessas sequências na base de dados do NCBI e o escore do alinhamento local e global das sequências, permitindo notar que determinadas sequências que serão utilizadas nos testes possuem um baixo grau de similaridade quando comparadas completamente, mas possuem um alto grau de similaridade ao considerar apenas uma de suas regiões. Também é possível verificar que algumas comparações de sequências grandes, tais como a 7M x 5M, podem ter escores locais muito pequenos (172) e comparações relativamente pequenas podem apresentar escores maiores, como a comparação 10K, que tem escore de alinhamento local igual a 5091. Os testes que serão apresentados neste capítulo foram feitos utilizando o alinhamento local das sequências.

5.3 Tempos de Execução e *Speedup*

Nessa seção, são relatadas diversas comparações entre resultados obtidos pelo Fickett-CUDAlign e três abordagens de obtenção do alinhamento local ótimo na comparação

das sequências biológicas: CUDAlign-Stage4-Optimized, CUDAlign-Stage4-Balanced e CUDAlign-Stage4-MM (Seção 3.2.4).

A Tabela 5.2 apresenta os tempos de execução dessas quatro diferentes soluções.

Tabela 5.2: Comparação entre os tempos de execução do Fickett-CUDAlign e três versões do CUDAlign. O tempo é aferido em milissegundos.

Comparação	Fickett-CUDAlign (ms)	CUDAlign-Stage4-Optimized (ms)	CUDAlign-Stage4-Balanced (ms)	CUDAlign-Stage4-MM (ms)
10k x 10k	**98,08**	179,62	227,95	284,70
57k x 57k	**0,74**	0,76	0,80	0,78
162k x 172k	0,83	0,82	**0,79**	1,14
543k x 536k	**1,96**	2,07	2,09	2,05
1M x 1M	**1403,09**	2555,49	3844,65	5072,56
3M x 3M	**109,69**	146,51	214,29	281,90
5M x 5M	**510,59**	26892,05	41958,09	54452,69
7M x 5M	**3,49**	4,84	5,26	9,23
10M x 10M	**898,65**	53563,24	81764,96	109509,22
23M x 25M	**10,15**	182,03	255,90	335,23
47M x 32M	**30425,82**	174147,98	255135,56	337737,63

Na Tabela 5.2 é possível notar que a estratégia Fickett-CUDAlign leva vantagem sobre as três abordagens de recuperação do alinhamento de sequências biológicas na grande maioria das comparações. Essa situação era esperada, haja vista o fato de que o Fickett-CUDAlign utiliza todas as otimizações das versões que foram com ele comparadas. Na Tabela 5.2, também se torna evidente que o tempo da estratégia CUDAlign-Stage4-Optimized é quase sempre melhor do que o tempo da estratégia CUDAlign-Stage4-Balanced e do CUDAlign-Stage4-MM, com três exceções. Na comparação 57k x 57k e 543k x 536k, os tempos de execução são superados pelo algoritmo CUDAlign-Stage4-MM e na comparação 162K x 172K o tempo de execução da estratégia CUDAlign-Stage4-Balanced é o menor de todos. Esses tempos serão analisados com detalhes em conjunto com o comportamento do alinhamento na Seção 5.5. A comparação do tempo de execução da versão CUDAlign-Stage4-Balanced com a versão CUDAlign-Stage4-MM indica que a versão CUDAlign-Stage4-Balanced é mais rápida, salvo as comparações 57K x 57K e 543k x 536k. Da mesma forma, isso será detalhado na Seção 5.5.

Nas comparações de sequências com tamanho a partir de 1M, existe um padrão claro no tempo de execução da recuperação do alinhamento. A estratégia Fickett-CUDAlign

possui sempre os melhores tempos de execução, seguida da estratégia CUDAlign-Stage4-Optimized, CUDAlign-Stage4-Balanced e, por último, a estratégia CUDAlign-Stage4-MM, o que indica que as otimizações que foram realizadas nas estratégias Fickett-CUDAlign, CUDAlign-Stage4-Optimized e CUDAlign-Stage4-Balanced, nessa ordem, são mais adaptadas para a recuperação de alinhamento de sequências longas.

A Tabela 5.3 apresenta o *speedup* da estratégia Fickett-CUDAlign em relação à execução das outras três abordagens. Nessa tabela, são apresentados os tempos do estágio 4 das estratégias comparadas.

Tabela 5.3: *Speedups* do Fickett-CUDAlign comparado ao Estágio 4 do CUDAlign.

Comparação	Fickett-CUDAlign vs CUDAlign-Stage4-Optimized	Fickett-CUDAlign vs CUDAlign-Stage4-Balanced	Fickett-CUDAlign vs CUDAlign-Stage4-MM	Escore Local
10k x 10k	1,83x	2,32x	2,90x	5091
57k x 57k	1,03x	1,08x	1,05x	52
162k x 172k	0,99x	0,95x	1,37x	18
543k x 536k	1,06x	1,07x	1,05x	48
1M x 1M	1,81x	2,74x	3,61x	88353
3M x 3M	1,34x	1,95x	2,57x	4226
5M x 5M	52,67x	82,18x	106,65x	5220960
7M x 5M	1,39x	1,51x	2,64x	172
10M x 10M	59,60x	90,99x	121,86x	10235188
23M x 25M	17,93x	25,21x	33,03x	9063
47M x 32M	5,72x	8,39x	11,10x	27206434

O *speedup* da estratégia Fickett-CUDAlign em relação à estratégia CUDAlign-Stage4-MM chega a 129,34 vezes na comparação de 10M x 10M. Já o *speedup* em relação à versão mais recente utilizada no CUDAlign (CUDAlign-Stage4-Optimized) chega a 59,60 vezes na comparação 10M x 10M, reduzindo o tempo de execução de 53,56 segundos para 0,90 segundo (Tabela 5.2).

Como pode ser visto na Tabela 5.3, as comparações que tiveram escore igual ou menor a 172 não apresentaram *speedup* expressivo: o maior *speedup* atingido dentro dessa faixa de valores foi 1,39x e o menor *speedup* foi de 0,99x em relação à abordagem CUDAlign-Stage4-Optimized. Sendo assim, nota-se que o tamanho do alinhamento tem grande impacto no *speedup* obtido pela estratégia Fickett-CUDAlign.

Além disso, é possível observar na execução do Fickett-CUDAlign que todos os alinhamentos com escore maior que 4.226 tiveram um ganho expressivo e que todos os grandes alinhamentos (que possuem escores na ordem de milhões) tiveram um *speedup*, ratificando a afirmação de que, em sequências com alinhamento grande o suficiente, o *speedup* médio tende a ser grande, ou seja, maior que 2,5x em relação à estratégia CUDAlign-Stage4-MM (Seção 4.2).

5.4 Estudo de *Speedup* do Fickett-CUDAlign

O Fickett-CUDAlign foi desenvolvido utilizando *Pthreads*, uma biblioteca que implementa a POSIX *Threads*, permitindo a execução paralela. Testes foram realizados para comparar o tempo de execução utilizando diversas quantidades de *threads* distintas. A Tabela 5.4 apresenta os tempos utilizados para a computação dos alinhamentos das sequências utilizando de 1 a 8 *threads* e a Figura 5.1 mostra o *speedup* obtido pela execução utilizando 8 *threads* em comparação com as execuções em 1, 2 e 4 *threads*.

Tabela 5.4: Tempos de execução do Fickett-CUDAlign em *threads*.

Comparação	1 *Thread*	2 *Threads*	4 *Threads*	8 *Threads*
10k x 10k	62.42	72.28	58.56	98.08
57k x 57k	0.52	1.77	1.71	0.74
162k x 172k	0.16	0.36	0.35	0.83
543k x 536k	0.61	1.16	1.15	1.96
1M x 1M	5390.70	2802.57	1581.82	1403.09
3M x 3M	143.72	112.80	109.93	109.69
5M x 5M	1102.19	796.73	586.03	510.59
7M x 5M	2.24	2.97	2.92	3.49
10M x 10M	1826.00	1177.02	993.47	898.65
23M x 25M	9.51	8.51	11.23	10.15
47M x 32M	107408.60	55886.16	35349.67	30425.82

Como esperado, o aumento de 4 para 8 *threads* não trouxe um benefício significativo, pois o processador possui apenas 4 núcleos reais. Os tempos de execução dependem do número de blocos iniciais. Por isso, no cálculo de alinhamentos muito pequenos o tempo de execução não é reduzido expressivamente quando acrescentamos *threads*.

A Figura 5.1 apresenta o *speedup* com 1, 2 e 4 *threads* para comparações de sequências que possuem escore maior que 50.000. Como pode ser visto na figura, os melhores *speedups* para 4 *threads* foram obtidos com as comparações 1M x 1M e 47M x 32M. Essas

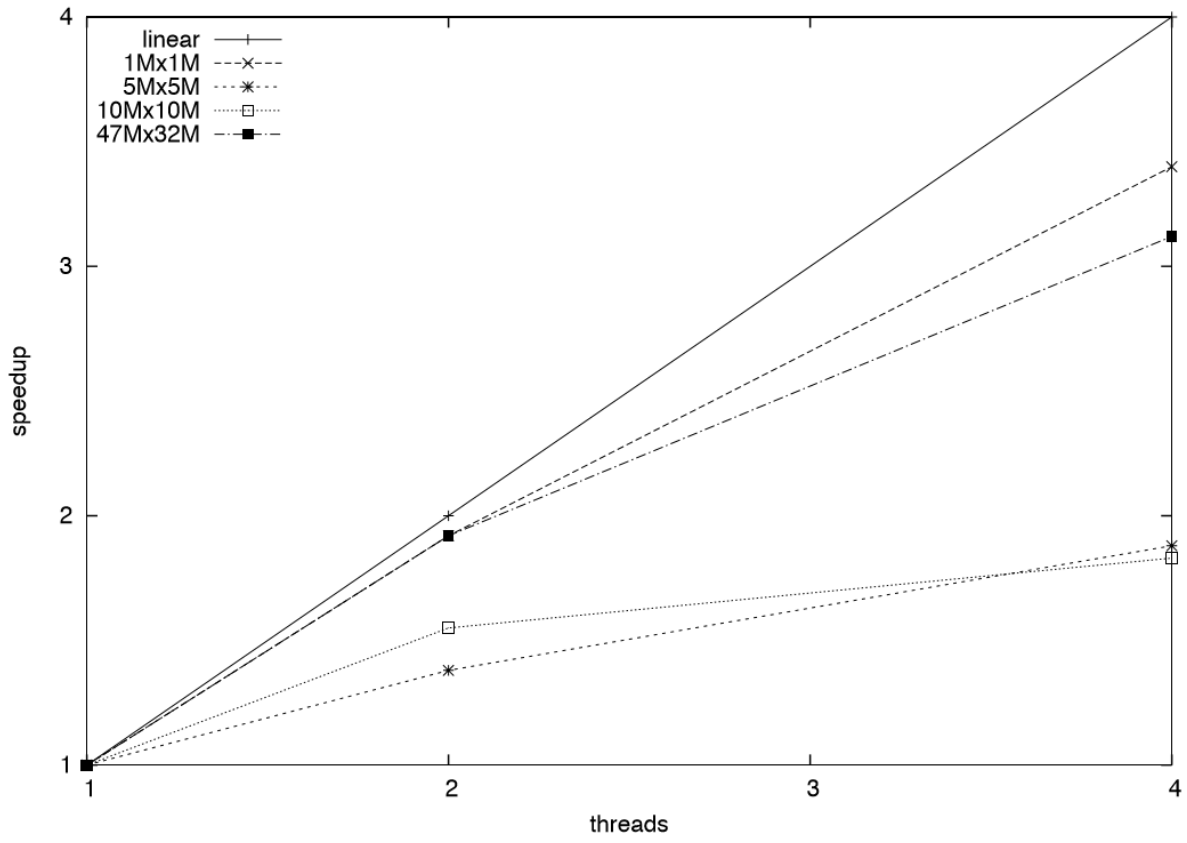


Figura 5.1: *Speedup* de execução do Fickett-CUDAlign de acordo com o número de *threads* utilizadas em diversos alinhamentos.

comparações possuem alinhamentos com um número significativo de *gaps* e *mismatches* e, aparentemente, isso contribui para que *speedup* seja obtido com múltiplas *threads*. As comparações 5M x 5M e 10M x 10M geram alinhamentos com alta taxa de *matches*. Acreditamos que isso faz com que as comparações sejam executadas muito rapidamente, não deixando espaço para melhorias do tempo de execução quando são utilizadas 4 *threads*. Acreditamos que o ganho com comparações de sequências maiores ($>$ 50.000.000) que gerem alinhamentos longos deve ser significativo.

5.5 Comportamento do Alinhamento

Para compreender melhor o comportamento dos alinhamentos, foi obtida a relação entre a quantidade de blocos e o tamanho percentual da faixa de uma dada comparação em relação ao tamanho total nos diversos blocos utilizados. Os resultados são apresentados na Tabela 5.5, que retrata o número de blocos onde o tamanho da faixa dividido pelo tamanho da subsequência é menor que uma porcentagem. Por exemplo, caso o tamanho da primeira subsequência seja 100 e o tamanho da faixa seja 8, temos 8% e o bloco é

Tabela 5.5: Número de blocos vs tamanho percentual da faixa.

Etapa	Relação entre tamanho da faixa e o tamanho da subsequência										Total de Blocos
	Até 10%	Até 20%	Até 30%	Até 40%	Até 50%	Até 60%	Até 70%	Até 80%	Até 90%	Até 100%	
10k x 10k	108	**241**	116	70	18	10	4	4	2	1	574
57k x 57k	0	**2**	1	0	0	0	0	0	0	0	3
162k x 172k	0	0	0	0	0	0	0	0	0	0	0
543k x 536k	0	1	**2**	0	0	0	0	0	0	0	3
1M x 1M	1056	3538	6469	**9940**	3615	2677	841	357	241	913	29.647
3M x 3M	**439**	179	74	69	60	59	38	58	65	164	1.205
5M x 5M	**323193**	119	28	25	17	8	8	9	9	52	323.468
7M x 5M	0	5	**14**	4	4	2	1	1	0	0	33
10M x 10M	**642225**	76	6	7	3	6	1	0	3	0	642.327
23M x 25M	**507**	3	1	0	0	0	0	0	0	0	511
47M x 32M	**1788870**	151556	25926	16259	6943	5670	4212	2724	2734	25079	2.029.973

contado na coluna "até 10%". Na comparação 5M x 5M, 323.193 blocos apresentaram faixas de até 10%, o que indica faixas bem pequenas e, portanto, um alto potencial de *speedup*. Os valores apresentados na Tabela 5.5 correspondem à totalidade dos blocos produzidos durante a execução da estratégia Fickett-CUDAlign, ou seja, durante todas as recursões do estágio 4.

Com base nos valores da Tabela 5.5, a compreensão dos *speedups* obtidos pelas sequências se torna mais simples, pois ela informa o quanto efetivamente foi deixado de ser computado na estratégia Fickett-CUDAlign. O fato de que uma faixa corresponde a até 10% do tamanho do bloco significa que os outros 90% das células não foram computados, gerando um *speedup* de pelo menos 10x em relação ao algoritmo de Myers-Miller (Seção 2.4). O fato de que uma faixa corresponde a 20% do bloco significa que os outros 80% não foram computados, gerando um *speedup* de pelo menos 5x, e assim sucessivamente até que a faixa corresponda ao tamanho do bloco (100%), onde o processamento do bloco irá ocorrer sem nenhuma redução, nesse caso não gerando nenhum *speedup* de processamento.

A comparação de 162k x 172k, em especial, não resultou em blocos, pois ela é tão pequena (escore 18) que foi executada de uma única vez. Por isso, não foi feito levantamento estatístico dessa sequência.

As comparações 5M x 5M, 10M x 10M e 23M x 25M foram as que obtiveram os *speedups* mais expressivos em comparação com o resultado obtido nas outras estratégias (Tabela 5.3). Como pode ser visto na Tabela 5.5, quase todos os valores das faixas de computação determinadas pela Equação (4.5) correspondem, no máximo, a 10% do tamanho total do bloco que foi calculado.

A comparação 1M x 1M possui uma configuração muito diferente das demais, pois a faixa percentual que possui mais blocos nessa comparação é a de que vai 30% a 40% com 9940 blocos.

A comparação 23M x 25M, embora apresente quase todas as faixas de computação com tamanhos que sejam de até 10% do bloco, não apresenta um *speedup* tão expressivo como

os *speedups* das comparações 10M x 10M e 5M x 5M. Nota-se, também, que a comparação 543K x 536K possui todos os blocos enquadrados antes da faixa de 50%. Porém ela não apresenta bom *speedup* (Tabela 5.3) conforme esperado. Para compreender esses valores, foi feita uma análise das faixas e do número de blocos iniciais que cada sequência possui (Tabela 5.6).

Tabela 5.6: Larguras das faixas e número de blocos iniciais das sequências utilizadas.

Comparação	Tamanho Mínimo	Tamanho Médio	Tamanho Máximo	Número de Blocos Iniciais
10K x 10K	1	12,00	696	2
57K x 57K	3	4,00	6	1
543K x 536K	5	6,00	9	1
1M x 1M	1	35,00	2815	61
3M x 3M	1	17,00	1433	2
5M x 5M	1	1,02	525	691
7M x 5M	2	13,50	80	1
10M x 10M	1	1,01	36	1257
23M x 25M	1	1,14	11	1
47M x 32M	1	6,50	31269	4513

A Tabela 5.6 apresenta o valor mínimo, médio e máximo das faixas computadas pela Equação (4.5). Desses valores observa-se que quase todas as comparações tiveram blocos que apresentaram o padrão *perfect_match*, pois o tamanho mínimo da faixa em quase todas as comparações foi 1. Os valores médios de tamanho das faixas nas comparações 1M x 1M e 3M x 3M foram os mais altos, ratificando o fato observado na Tabela 5.5, de que essas sequências possuem blocos em todas as faixas.

O número de blocos iniciais das comparações 57K x 57K, 543K x 536K, 7M x 5M e 23M x 25M é 1, o que irá resultar no retardo do paralelismo do processamento dessas comparações. Essa informação explica o motivo pelo qual as comparações 543K x 536K e 23M x 25M não tiveram um desempenho tão bom quanto o esperado pela redução da área a ser computada, pois embora as sequências tenham margem para a redução da área a ser computada, as diversas *threads* criadas para a execução do Algoritmo 1 (Seção 4.4) aguardam a criação de novos blocos através de novas camadas recursivas sem que possam executar até que os blocos sejam criados.

Para verificar se os números obtidos na Tabela 5.5 se aproximam do valor da distribuição real de *gaps* que cada alinhamento apresenta, foram realizados novos testes. A

Tabela 5.7 apresenta o tamanho do alinhamento de cada uma das sequências utilizadas e seus respectivos escores. Além disso, a Tabela 5.7 também informa o percentual de *matches*, *mismatches* e *gaps* que cada alinhamento possui.

Tabela 5.7: Escore, tamanho do alinhamento e percentuais de *matches*, *mismatches* e *gaps* das sequências utilizadas.

Comparação	Tamanho do Alinhamento	Escore	*Matches* (%)	*Mismatches* (%)	*Gaps* (%)
10K x 10K	9124	5091	89,12	9,64	1,24
57K x 57K	80	52	92,50	5,00	2,50
162K x 172K	18	18	100,00	0,00	0,00
543K x 536K	92	48	88,04	11,96	0,00
1M x 1M	472249	88353	79,76	17,12	3,12
3M x 3M	14582	4226	83,05	10,46	6,49
5M x 5M	5229192	5220960	99,95	0,00	0,05
7M x 5M	565	172	84,07	12,74	3,19
10M x 10M	10236796	10235188	99,99	0,00	0,00
23M x 25M	9107	9063	99,88	0,05	0,07
47M x 32M	33583690	27206434	94,38	1,54	4,08

Na Tabela 5.7, é possível confirmar os dados da Tabela 5.5, pois as comparações 5M x 5M e 10M x 10M apresentaram a menor quantidade percentual de *gaps* e as comparações 10K x 10K, 1M x 1M, 3M x 3M, e 47M x 42M estão entre as que possuem as maiores quantidades percentuais de *gaps* contabilizados.

Há que se observar ainda que não é apenas o número de *gaps* que interfere no tamanho da faixa que será utilizada, mas também o número de *mismatches*, pois eles causam um falso positivo na estratégia Fickett-CUDAlign, conforme visto na Seção 4.2. E nota-se que quanto maior o percentual de *matches* ocorridos no alinhamento (Tabela 5.7), menor foi o percentual de *gaps* encontrado e mais blocos dessa comparação estão nas colunas de percentuais mais baixos de tamanho da faixa em relação ao tamanho total do bloco na Tabela 5.5.

Após todas as coletas de dados realizada nessa seção foram levantadas informações suficientes para explicar o comportamento do algoritmo Fickett-CUDAlign de modo genérico para diversos comportamentos de alinhamentos.

No caso de comparações que possuam um alinhamento com grau médio ou baixo de similaridade como a comparação 1M x 1M é esperado um gráfico de distribuição de blocos semelhante ao da Figura 5.2. Nessa figura a faixa percentual com o maior número

de blocos fica próxima do meio, mostrando que uma quantidade grande de blocos fica distante do $perfect\_match$ mas que ainda sim é possível obter um *speedup* significativo utilizando as faixas de computação.

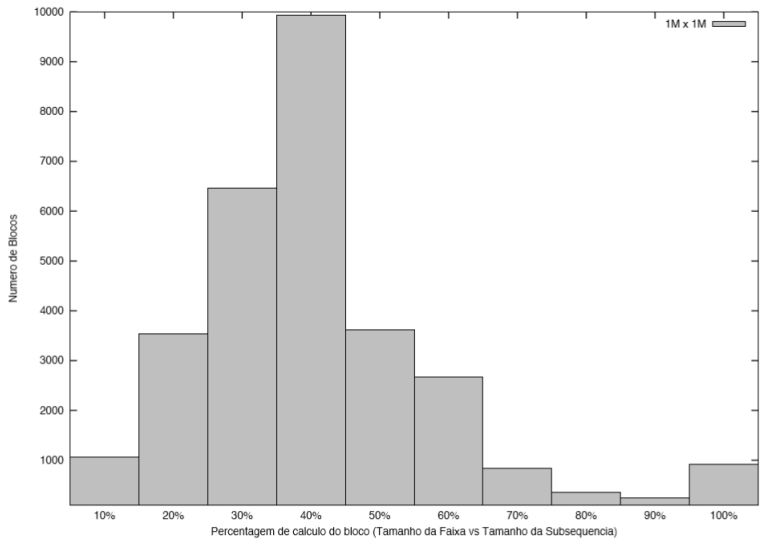


Figura 5.2: Número de blocos em cada faixa percentual da comparação 1M x 1M.

Já no caso de comparações que tenham um alinhamento com grande similaridade como ocorre na comparação 10M x 10M existe um padrão semelhante ao da Figura 5.3, onde praticamente todos os blocos ficam na faixa de até 10%. Isso mostra um grande potencial para *speedup* com o uso de faixas de computação nesse tipo de comparação.

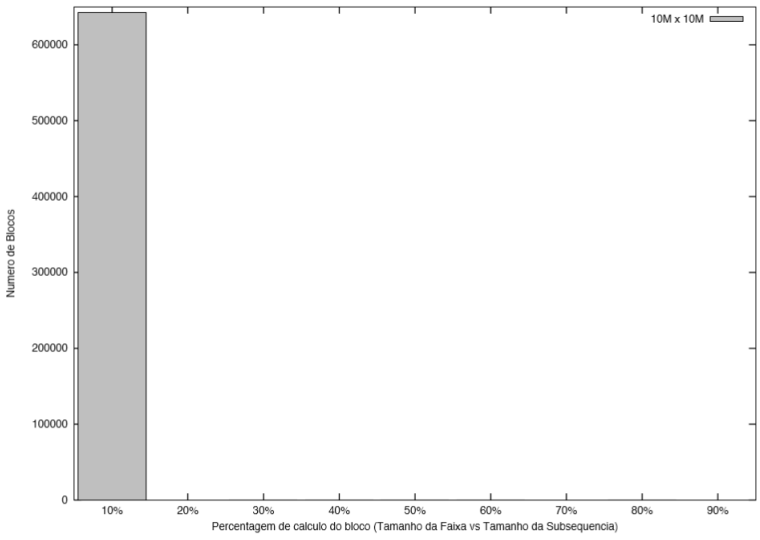


Figura 5.3: Número de blocos em cada faixa percentual da comparação 10M x 10M.

Outras comparações são combinações desses padrões e portanto têm um ganho de desempenho que fica entre os dois modelos apresentados. A Figura 5.4 é um exemplo de

uma comparação que possui um alinhamento que fica entre os dois padrões apresentados, e também mostra possibilidade de se obter um *speedup* significativo ao utilizar faixas de computação.

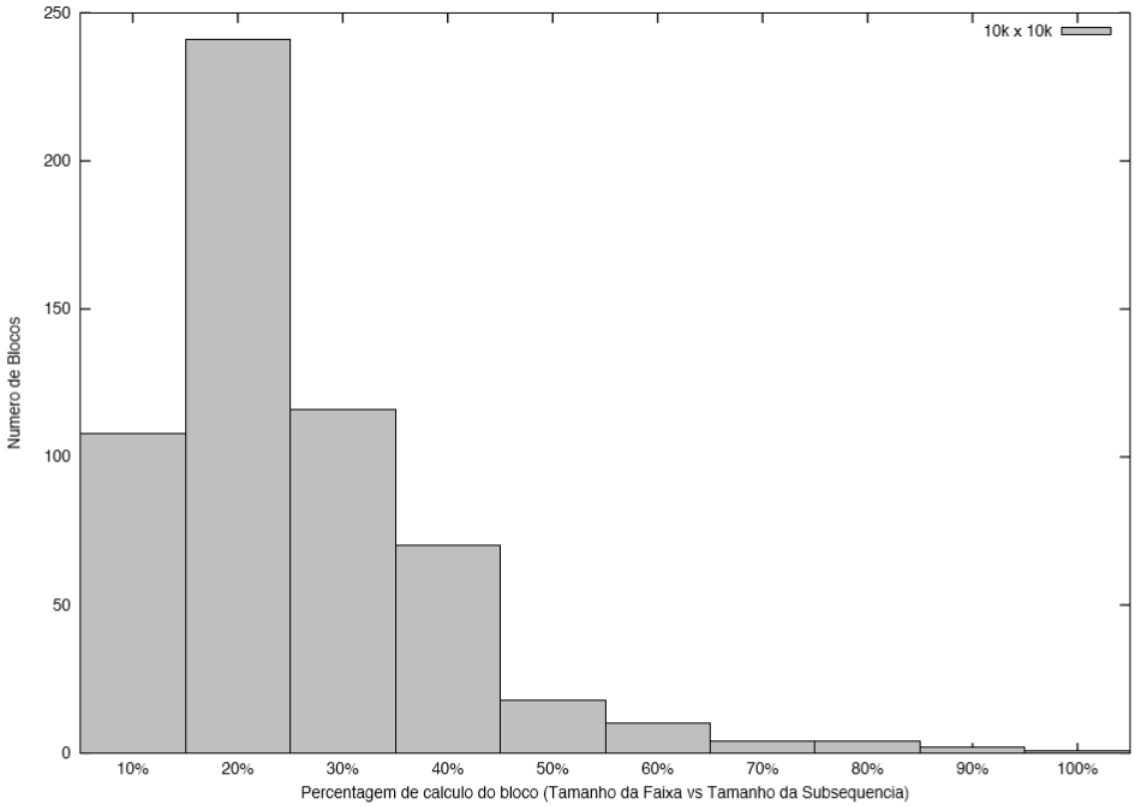


Figura 5.4: Número de blocos em cada faixa percentual da comparação 10K x 10K.

5.6 Comparação entre o Fickett-CUDAlign e o SW#

Para obter um comparativo entre o Fickett-CUDAlign e outra ferramenta que utilize GPU para execução do algoritmo SW, foi escolhida a ferramenta SW# (Seção 3.2.3), pois ela é otimizada para comparações de sequências longas (Seção 3.2.3), assim como o Fickett-CUDAlign. A Tabela 5.8 apresenta um comparativo entre o tempo total de execução das ferramentas Fickett-CUDAlign, o SW# e o CUDAlign4.0, todos utilizando o ambiente computacional descrito na Seção 5.1.

Tabela 5.8: Tempos de execução e *speedups* entre o Fickett-CUDAlign e o SW#.

Comparação	Fickett-CUDAlign	SW#	Fickett-CUDAlign vs SW#	CUDAlign4.0
10k x 10k	0,16	0,83	**5,19x**	0,3
57k x 57k	0,17	0,89	**5,23x**	0,17
162k x 172k	0,98	1,57	1,60x	0,98
543k x 536k	0,56	8,18	**14,60x**	0,56
1M x 1M	27,72	45,10	1,63x	28,86
3M x 3M	229,13	249,70	1,09x	229,16
5M x 5M	330,73	590,82	1,79x	357,11
7M x 5M	823,59	896,81	1,09x	823,59
10M x 10M	1244,55	2223,73	1,79x	1297,22
23M x 25M	12387,06	13498,00	1,09x	12387,23
47M x 32M	25383,99	48767,00	1,92x	25691,30

A partir dos dados apresentados na Tabela 5.8, é possível observar que as comparações que obtiveram os melhores *speedups* do CUDAlign em relação ao SW# foram as comparações 10K x 10K, 57K x 57K e 543K x 536K. Isso ocorre porque, apesar de nenhuma das ferramentas ser voltada para a realização de pequenos alinhamentos, pelos resultados apresentados nota-se que o Fickett-CUDAlign possui melhor desempenho para alinhamentos pequenos.

Nas comparações de sequências de tamanho 1M ou maior, o Fickett-CUDAlign possui *speedups* menores em relação ao SW#, mas ainda assim é mais rápido em todos os testes realizados.

Em termos absolutos, o tempo de execução da comparação 47M x 32M foi reduzido de 13 horas e 32 minutos (SW#) para 7 horas e 3 minutos (Fickett-CUDAlign).

Embora os tempos de execução do Fickett-CUDAlign e do CUDAlign4.0 sejam muito próximos, é possível notar que as comparações 10K x 10K, 1M x 1M, 5M x 5M e 10M x 10M tiveram bons ganhos na execução na comparação entre o Fickett-CUDAlign e o CUDAlign4.0, mesmo ciente de que o algoritmo Fickett-CUDAlign está sendo aplicado apenas no estágio 4 da execução do CUDAlign. Isso mostra que as otimizações realizadas na ferramenta através da estratégia Fickett-CUDAlign auxiliaram na obtenção de resultados de alto desempenho, como pode ser visto na Tabela 5.8. Acredita-se que esses ganhos sejam ainda mais expressivos para comparações de sequências maiores (e.g., 100MB), com múltiplas GPUs.

Como já fora analisado nos testes realizados nesse capítulo, as sequências utilizadas para comparação variam do *perfect match* à baixa similaridade entre as sequências, e outras sequências que poderiam ser comparadas consistiriam em apenas combinações dos comportamentos de alinhamentos já testados. Isso nos leva a crer que é possível generalizar o ganho de desempenho obtido pelo Fickett-CUDAlign em relação a outras abordagens e ferramentas para a recuperação do alinhamento ótimo caso outras sequências fossem utilizadas.

Capítulo 6

Conclusão e Trabalhos Futuros

6.1 Conclusão

Na presente dissertação, foi proposta e avaliada uma solução paralela de comparação exata de sequências genômicas visando reduzir o número de células calculadas na matriz de programação dinâmica e, consequentemente, aumentar o desempenho da comparação. Esta solucão, nomeada Fickett-CUDAlign, acelera a computação de blocos na matriz, delimitados por pontos chamados *crosspoints*. De posse de dois *crosspoints*, a estratégia Fickett-CUDAlign faz uma adaptação do algoritmo de Fickett (Seção 2.5), combinando-o com o algoritmo de Myers-Miller (Seção 2.4).

No projeto da estratégia (Capítulo 4) foi desenvolvida a Equação (4.5), que permite calcular o tamanho da faixa de Fickett que será utilizada em cada bloco, de modo que o alinhamento esteja sempre entre os limites da faixa, pois na proposta original de Fickett [9] não foi desenvolvida uma maneira de calcular o tamanho da faixa, assumindo até que a faixa não englobe todo o alinhamento e seja necessário recalculá-lo.

Dentre as contribuições da estratégia Fickett-CUDAlign, destaca-se que ela pode ser aplicada em algoritmos recursivos de obtenção do alinhamento, como é o caso do algoritmo de Myers-Miller. Isso faz com que a estratégia Fickett-CUDAlign utilize menos memória e possa computar rapidamente alinhamentos de sequências maiores. Além disso, diferentemente do algoritmo de Fickett, que utiliza um tamanho de faixa para delimitar a área a ser computada em toda a comparação, a estratégia Fickett-CUDAlign utiliza faixas de tamanho variável e ajustável para computar o alinhamento, permitindo que a faixa assuma valores pequenos onde o alinhamento se aproxima do *perfect match* e valores grandes onde a quantidade de *gaps* é maior, isto é, Fickett-CUDAlign é uma estratégia adaptativa.

Durante a fase de testes (Capítulo 5.5) foram observados os comportamentos de diversas comparações de sequências, que variavam do tamanho 10K x 10K a 47M x 32M.

Foi possível observar que, nesses casos, o desempenho da ferramenta Fickett-CUDAlign é melhor do que as outras abordagens anteriormente utilizadas na ferramenta CUDAlign. Em comparação com a estratégia CUDAlign-Stage4-Optimized (Seção 3.2.4), o Fickett-CUDAlign chegou a obter um *speedup* de 59,60 vezes na comparação 10M x 10M, reduzindo o tempo de execução de 53,56 segundos para 0,90 segundo. Em relação à estratégia CUDAlign-Stage4-MM, obteve-se um *speedup* de 121,86 vezes na mesma comparação e o tempo foi reduzido de 1 minuto e 49,51 segundos para 0,90 segundo.

6.2 Trabalhos Futuros

Como trabalhos futuros sugerimos:

- **Integração do Fickett-CUDAlign ao estágio 3 do CUDAlign:** Atualmente o Fickett-CUDAlign é executado no estágio 4 em CPU. Para que maiores *speedups* sejam alcançados é proposto, como trabalho futuro, a integração da estratégia Fickett-CUDAlign ao estágio 3, criando uma versão do Fickett-CUDAlign para GPU. Com isso, esperamos que os *speedups* obtidos pela estratégia tenham mais impacto no tempo total de computação do alinhamento.

  Para que o Fickett-CUDAlign, que foi projetado para o estágio 4 com processamento realizado em CPU, possa ser aplicado ao estágio 3 em GPU, é desejável que o Fickett-CUDAlign utilize a técnica de *wavefront* para realizar as comparações de sequências. Esta técnica confere maior paralelismo ao alinhamento das sequências devido à independência dos valores das células das anti-diagonais, assim, permite explorar melhor o paralelismo do processamento das GPUs, que possuem milhares de núcleos de processamento disponíveis para realizar as comparações.

  Além disso, para utilizar a estratégia Fickett-CUDAlign no estágio 3 também seria necessário aplicar a técnica *cells delegation* (Seção 3.2.4) para explorar o paralelismo da técnica de *wavefront*. Desse modo, os formatos de blocos de processamento do Fickett-CUDAlign apresentados nos capítulos 4 e 5 seriam substituídos por formatos de paralelogramos de processamento quando a estratégia Fickett-CUDAlign fosse empregada na GPU.

- **Processamento com GPU do estágio 4:** Sugere-se também que seja utilizada a GPU na computação do estágio 4 do CUDAlign. Nas comparações que possuem alinhamentos grandes, muitos *crosspoints* iniciais foram gerados, chegando a 4513 no caso da comparação 47M x 32M. Esses *crosspoints* poderiam ser computados em paralelo caso uma GPU seja utilizada, acelerando a obtenção de resultados.

Para colocar o estágio 4 em GPU seria necessário utilizar as técnicas de *wavefront* e *cells delegation* da mesma forma como ocorreria no estágio 3, colocando diferentes diagonais para serem computadas em *threads* distintas.

Após as modificações do processamento do estágio 4 com GPU e a utilização do Fickett-CUDAlign no estágio 3, a CPU seria utilizada somente nos estágios 5 e 6. Esses estágios têm um impacto pouco significativo no desempenho do Fickett-CUDAlign, sendo que o estágio 6 é um estágio opcional e o estágio 5 é executado tão rapidamente na CPU que a utilização da GPU não seria recomendada, devido à falta de paralelismo.

- **Comparação de outros tipos de sequências biológicas:** A estratégia CUDAlign foi projetada apenas em comparações de sequências de DNA. Contudo, as sequências de RNA e as sequências protéicas também poderiam ser computadas com essa estratégia, estendendo o seu escopo. Para isso seria necessário acrescentar os caracteres que não fazem parte do alinhamento de nucleotídeos ao alfabeto da comparação.

  Além disso, como as sequências protéicas são menores que as sequências de nucleotídeos, seria necessário realizar mais de uma comparação de sequência, paralelamente, para que seja possível obter um *speedup* significativo na comparação de sequências de proteínas. Isso é muito interessante na computação de comparações de proteínas, pois a maior parte dessas comparações é realizada entre uma sequências e um banco de dados protéico, permitindo o paralelismo necessário para a execução na GPU.

- **Integração com outras estratégias:** O Fickett-CUDAlign poderia ser integrado com outras estratégias, como o IST (Seção 3.2.4), de modo a aumentar o *speedup* já obtido. Nessa situação o Fickett-CUDAlign serviria para acelerar a computação do bloco após ser especulado um valor para os escores do alinhamento e, dessa forma, poder-se-ia determinar os valores do comprimento da faixa depois da computação da equação descrita na Seção 4.2, acelerando o processamento.

- **Integração com outras ferramentas:** Sugere-se a integração da estratégia Fickett-CUDAlign à ferramenta SW# (Seção 3.2.3), que implementa o algoritmo Myers-Miller original. Acreditamos que as faixas múltiplas de tamanho ajustável vão proporcionar ganhos significativos na execução dessa ferramenta, porque após o término da primeira etapa do algoritmo de Myers-Miller o Fickett-CUDAlign já poderia ser executado da mesmo forma como foi executado no CUDAlign.

- **Ajuste do número de *threads* com o número de blocos do alinhamento:** Pode-se verificar nas comparações realizadas, uma grande variação no número de

blocos que cada alinhamento é capaz de produzir, com quantidades variando de um a dezenas de milhões. As comparações que possuem poucos blocos são menos passíveis de serem paralelizadas. Para utilizar as *threads* de maneira a obter o melhor desempenho, sugere-se que o número de *threads* utilizadas seja ajustável de acordo com o número de blocos do alinhamento.

Referências

[1] Amos Bairoch and Brigitte Boeckmann. The swiss-prot protein sequence data bank. *Nucleic Acids Research*, 19(suppl):2247–2249, 1991. 15, 16, 19, 20

[2] R. Batista, A. Boukerche, and A. Demelo. A parallel strategy for biological sequence alignment in restricted memory space. *Journal of Parallel and Distributed Computing*, 68(4):548–561, April 2008. 14

[3] Azzedine Boukerche, Rodolfo Bezerra Batista, Alba Cristina Magalhaes Alves De Melo, Felipe Brandt Scarel, and Lavir Antonio Bahia Carvalho De Souza. Exact parallel alignment of megabase genomic sequences with tunable work distribution. *International Journal of Foundations of Computer Science*, 23(02):407–429, 2012. 14

[4] Azzedine Boukerche, Alba Cristina Magalhaes Alves de Melo, Edans Flavius de Oliveira Sandes, and Mauricio Ayala-Rincón. An exact parallel algorithm to compare very long biological sequences in clusters of workstations. *Cluster Computing*, 10(2):187–202, 2007. 2

[5] Edans F. de O. Sandes, Guillermo Miranda, Alba C.M.A. de Melo, Xavier Martorell, and Eduard Ayguade. Cudalign 3.0: Parallel biological sequence comparison in large gpu clusters. *2014 14th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing*, 0:160–169, 2014. x, 2, 31, 32

[6] Edans Flavius de O. Sandes and Alba Cristina M.A. de Melo. Retrieving smith-waterman alignments with optimizations for megabase biological sequences using gpu. *IEEE Transactions on Parallel and Distributed Systems*, 24(5):1009–1021, 2013. 2, 3, 16, 31

[7] E.F. de O Sandes and A.C.M.A. de Melo. Smith-waterman alignment of huge sequences with gpu in linear space. In *Parallel Distributed Processing Symposium (IPDPS), 2011 IEEE International*, pages 1199–1211, May 2011. ix, x, 25, 27, 28, 30

[8] Michael Farrar. Striped smith–waterman speeds database searches six times over other simd implementations. *Bioinformatics*, 23(2):156–161, January 2007. 2, 15

[9] J. W. Fickett. Fast optimal alignment. *Nucleic acids research*, 12(1 Pt 1):175–179, January 1984. ix, 2, 10, 11, 66

[10] M. J. Flynn. Some Computer Organizations and Their Effectiveness. *IEEE Transactions on Computers*, C-21(9):948–960, September 1972. 17

[11] Osamu Gotoh. An improved algorithm for matching biological sequences. *Journal of molecular biology*, 162(3):705–708, December 1982. 8, 9

[12] P. Green. SWAT Optimization. http://www.phrap.org/phredphrap/swat.html, 1993. 15

[13] Dan Gusfield. *Algorithms on strings, trees, and sequences : computer science and computational biology*. Cambridge Univ. Press, January 2007. 1, 4, 10

[14] Daniel S. Hirschberg. Algorithms for the Longest Common Subsequence Problem. *J. ACM*, 24(4):664–675, October 1977. 2, 9

[15] Fumihiko Ino, Yuma Munekawa, and Kenichi Hagihara. Sequence homology search using fine grained cycle sharing of idle gpus. *IEEE Trans. Parallel Distrib. Syst.*, 23(4):751–759, 2012. 2

[16] Xianyang Jiang, Xinchun Liu, Lin Xu, Peiheng Zhang, and Ninghui Sun. A reconfigurable accelerator for smith-waterman algorithm. *Circuits and Systems II: Express Briefs, IEEE Transactions on*, 54(12):1077–1081, Dec 2007. 2

[17] Matija Korpar and Mile Šikić. SW#–GPU-enabled exact alignments on genome scale. *Bioinformatics*, 29(19):2494–2495, October 2013. ix, 21, 22

[18] Yongchao Liu, Douglas L. Maskell, and Bertil Schmidt. CUDASW++: optimizing Smith-Waterman sequence database searches for CUDA-enabled graphics processing units. *BMC research notes*, 2(1):73+, 2009. 19

[19] Yongchao Liu, Bertil Schmidt, and Douglas Maskell. CUDASW++2.0: enhanced Smith-Waterman protein database search on CUDA-enabled GPUs based on SIMT and virtualized SIMD abstractions. *BMC Research Notes*, 3(1):93+, 2010. 2, 20

[20] Yongchao Liu, Adrianto Wirawan, and Bertil Schmidt. CUDASW++ 3.0: accelerating Smith-Waterman protein database search by coupling CPU and GPU SIMD instructions. *BMC Bioinformatics*, 14(1):117+, April 2013. 20

[21] David Mount. *Bioinformatics: Sequence and Genome Analysis*. Cold Spring Harbor Laboratory Press, 2nd edition, March 2013. 1, 4, 5, 6, 8

[22] E. W. Myers and W. Miller. Optimal alignments in linear space. *Computer applications in the biosciences : CABIOS*, 4(1):11–17, March 1988. ix, 2, 9, 10, 50

[23] S. B. Needleman and C. D. Wunsch. A general method applicable to the search for similarities in the amino acid sequence of two proteins. *Journal of molecular biology*, 48(3):443–453, March 1970. 5, 6

[24] NVidia. CPU vs GPU performance. http://www.nvidia.com.br. Acessado em Fevereiro de 2016. ix, 16, 17, 18

[25] NVidia. Plataforma de computação paralela. http://www.nvidia.com.br/object/cuda_home_new_br.html. Acessado em Fevereiro de 2016. 17

[26] S. Rajko and S. Aluru. Space and time optimal parallel sequence alignments. *Parallel and Distributed Systems, IEEE Transactions on*, 15(12):1070–1081, Dec 2004. 13, 14

[27] T. Rognes and E. Seeberg. Six-fold speed-up of Smith-Waterman sequence database searches using parallel processing on common microprocessors. *Bioinformatics (Oxford, England)*, 16(8):699–706, August 2000. 2, 15

[28] Edans Flavius O. Sandes and Alba Cristina M.A. de Melo. Cudalign: Using gpu to accelerate the comparison of megabase genomic sequences. In *Proceedings of the 15th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '10, pages 137–146, New York, NY, USA, 2010. ACM. ix, 2, 22, 23, 24, 29

[29] E.F.O. Sandes and A.C.M.A. Melo. Algoritmos paralelos exatos e otimizações para alinhamento de sequências biológicas longas em plataformas de alto desempenho. Universidade de Brasília, Tese de Doutorado, 2015. 29

[30] E.F.O. Sandes, G. Miranda, X. Martorell, E. Ayguade, G. Teodoro, and A.C.M.A. Melo. Cudalign 4.0: Incremental speculative traceback for exact chromosome-wide alignment in gpu clusters. *Parallel and Distributed Systems, IEEE Transactions on*, PP(99):1–1, 2016. ix, 2, 25, 32, 33

[31] M.C. Schatz and B. Langmead. The dna data deluge. *Spectrum, IEEE*, 50(7):28–33, July 2013. 1

[32] Carlos Setubal and Joao Meidanis. *Introduction to Computational Molecular Biology*. PWS Publishing, 1 edition, January 1997. 4, 7

[33] T. F. Smith and M. S. Waterman. Identification of common molecular subsequences. *Journal of Molecular Biology*, 147(1):195–197, March 1981. 1, 7