

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

Checkpoint local e não coordenado para a aplicação paralela MASA-CUDAlign de comparação de sequências biológicas

Isabel P. Starling
Gustavo L. Dezan

Monografia apresentada como requisito parcial
para conclusão do Bacharelado em Ciência da Computação

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

Prof.a Dr.a Carla Denise Castanho
CIC/UnB

Prof. Dr. Vinicius Ruela Pereira Borges
CIC/UnB

Prof. Dr. Marcelo Grandi Mandelli
Coordenador do Bacharelado em Ciência da Computação

Brasília, 04 de dezembro de 2025

Agradecimentos

Eu, Isabel Pierre Starling, chego ao término desta jornada acadêmica contemplando não uma conquista individual, mas o ápice de um esforço coletivo. É, portanto, com profunda gratidão que registro meu reconhecimento a todas as pessoas que foram fundamentais para a concretização deste trabalho.

Agradeço aos meus pais, Fernando Starling e Denise Pierre, pela educação e por sempre priorizarem minha formação. Obrigada por criarem um ambiente onde meus estudos puderam ser o foco.

À Professora Alba, minha sincera gratidão pela disponibilidade, pela orientação e pelo apoio constante durante todo o desenvolvimento deste projeto.

Ao Gustavo Dezan, pelo empenho, colaboração e trabalho em equipe durante o desenvolvimento do projeto.

Ao grupo de programação competitiva da UnB, Unballoon, o meu agradecimento por ter me mostrado e ensinado programação competitiva. Foi nesse ambiente que descobri minha paixão por essa área.

Ao meu time de maratona, A Turma da Hello Kitty, Bruno Vargas e Enzo Yoshio, é uma honra fazer parte desta equipe. Agradeço por todas as horas de treinamento, pelas competições, pela parceria e, sobretudo, por acreditarem em mim e me motivarem a sempre buscar aprender mais e melhorar.

Aos meus amigos, que tornaram essa trajetória acadêmica muito mais leve. Obrigada pela cumplicidade, pelos momentos de descontração e pelo apoio nos períodos mais tensos.

Ao meu companheiro, Luiz Carlos da Silva Néto Vartuli, meu agradecimento mais especial por estar ao meu lado nesta reta final. Obrigada por todo o carinho, por sempre me ouvir e pelo suporte emocional que você me deu. Sua presença fez toda a diferença.

Por fim, à Universidade de Brasília e a todos os professores que contribuíram para minha formação ao longo destes anos. Esta conquista é também um reflexo do conhecimento que aqui pude construir.

Eu, Gustavo Lopes Dezan, diante desse momento que marca não somente o final de uma caminhada, mas principalmente a transição para uma nova jornada, coloco em palavras agradecimentos muito devidos, pois um caminho não se trilha sozinho e o trajeto é, muitas vezes, tão importante quanto seu destino.

Agradeço a minha família, meus pais Myrthes D. e Sandro D., e meu irmão Matheus D., pelo incomparável apoio.

Agradeço ao Elson P., Suzana R., e Eliane F., por serem minha segunda casa.

Aos amigos que caminharam ao meu lado, Alvaro S., Arthur B., Breno R., Daniel T., Davi S., Eduardo B., Felipe M., Gabriel M., Gustavo Rafael B., Jordana M., Lucas N., Nathalia, Pedro Z., Rafael P., Sávio V., Vitor L., Waldes C., por compartilharem comigo os bons momentos e os aprendizados de uma vida inteira.

Aos amigos da Schief.ai, Adriana G., Beatriz A., Carlos A., Cassiano T., Davi S., Eduardo M., Eduardo S., Fábio F., Gustavo S., João D., João Victor A., Lucas A., Lucas H., Marcelo John F., Marco P., Matheus D., Murillo P., Paula F., Rodrigo L., Rodrigo Z., Vinicius O., Walter A., pelo ambiente amigável, pela excelência e por criarmos algo juntos!

À Isabel S., pela dedicação e proatividade e por todo o esforço empenhado no desenvolvimento do trabalho aqui apresentado.

À Professora Alba M., pela atenção e orientação, bem como pela admirável dedicação com que desempenha a profissão de educadora.

O presente trabalho foi realizado com apoio da Coordenação de Aperfeiçoamento de Pessoal de Nível Superior - Brasil (CAPES), por meio do Acesso ao Portal de Periódicos.

Resumo

Este trabalho apresenta um mecanismo de *checkpoint* local e não coordenado para o FT-CUDAlign, ferramenta tolerante a falhas derivada do MASA-CUDAlign para alinhamento exato de sequências biológicas em *GPUs*. Diferentemente do método tradicional baseado apenas em *checkpoints* globais tomados nos breakpoints de cada iteração, a proposta permite que cada GPU salve periodicamente a última linha concluída de sua região da matriz, reduzindo significativamente o retrabalho após falhas. Os checkpoints são armazenados em formato binário, utilizando operações atômicas de escrita e validação imediata por desserialização, garantindo a integridade dos arquivos e mantendo apenas estados válidos. Os resultados experimentais mostram que a sobrecarga introduzida é mínima, preservando o desempenho da ferramenta. Além disso, este trabalho integra ao FT-CUDAlign a atualização que substitui a memória de textura por memória global, eliminando a limitação de 134 MB que impedia o processamento eficiente de sequências longas. Esta mudança estrutural elimina a necessidade de particionamento e melhora o acesso aos dados, beneficiando tanto a execução normal quanto a recuperação pós-falha. Como trabalhos futuros, destaca-se a extensão do mecanismo de *checkpoint* local para execuções em múltiplas *GPUs*, a implementação da recuperação a partir desses *checkpoints* e a criação de um parâmetro dedicado para seu gerenciamento, visando uma tolerância a falhas mais granular e eficiente em ambientes distribuídos.

Palavras-chave: Checkpoints, tolerância a falhas, comparação de sequências, GPU

Abstract

This work presents a local, uncoordinated checkpointing mechanism for FT-CUDAlign, a fault-tolerant extension of the MASA-CUDAlign framework for exact biological sequence alignment on GPUs. While FT-CUDAlign traditionally relies solely on global application-level checkpoints taken at predefined breakpoints, this approach can lead to substantial recomputation when failures occur between iterations. The method proposed here enables each GPU to periodically save the most recently completed alignment row of its local matrix region, producing lightweight, fine-grained recovery points without requiring synchronization across devices. These checkpoints are written in binary format using atomic file operations and immediate deserialization for integrity validation, ensuring that only consistent states are retained. Experimental results show that the mechanism introduces minimal overhead, while integrating transparently into the existing execution flow. Additionally, this work updates FT-CUDAlign by replacing texture memory with global memory for sequence storage, removing the 134 MB size limitation that previously forced inefficient matrix partitioning and improving memory-access efficiency during both normal execution and recovery. Future work includes extending local checkpointing to multi-GPU executions, implementing full recovery from saved checkpoints, and introducing a dedicated parameter for fine-grained checkpoint management, enabling more efficient and granular fault tolerance in large-scale distributed environments.

Keywords: Checkpoints, fault tolerance, sequence comparison, GPU

Sumário

1 Introdução **1**

2 Comparacao de Sequencias Biologicas **3**

2.1 Visão Geral 3

2.1.1 Pontuação dos Alinhamentos 4

2.1.2 Alinhamento Global e Local 4

2.1.3 Programação Dinâmica 5

2.1.4 Algoritmos Exatos 5

2.2 Algoritmo Smith-Waterman 6

2.2.1 Construção da Matriz de Alinhamento Smith-Waterman 6

2.2.2 Recuperação do Alinhamento Smith Waterman Ótimo (Traceback) 7

2.3 Algoritmo Gotoh 8

2.3.1 Construção da Matriz de Alinhamento Gotoh 8

2.3.2 Recuperação do Alinhamento Ótimo (*Traceback*) Gotoh 10

2.4 Algoritmo Myers-Miller 11

2.4.1 Construção da Matriz de Alinhamento Myers-Miller 11

2.4.2 Recuperação do Alinhamento Ótimo (*Traceback*) Myers-Miller . . . 12

3 Evolução da Ferramenta *MASA-CUDAlign* **14**

3.1 *MASA-CUDAlign* 14

3.1.1 Block Pruning 17

3.1.2 *CUDAlign Multi-GPU* 18

3.2 *DynamicBP* 19

3.3 *FT-CUDAlign* 20

3.3.1 Detecção Pré-Execução 21

3.3.2 Detecção Durante a Execução 22

3.3.3 Recuperação 22

3.3.4 Protocolos de Término 22

4 Projeto do *Checkpoint* Local para o *FT-CUDAlign* **24**
4.1 Visão geral 24
4.2 Tomada de *Checkpoint* local 26
4.3 Projeto do *checkpoint* local 27
4.4 Memória de textura 29

5 Resultados Experimentais **30**
5.1 Ambiente de Testes 30
5.2 Resultados da Execução 31
5.3 Teste do *Checkpoint* 32

6 Conclusão **33**

Referências **35**

Anexo **36**

I Artigo apresentado no WPOS 2025 **37**

Lista de Figuras

2.1 Alinhamento global e local entre duas sequências biológicas. 5
2.2 Matriz do Smith Waterman 7
2.3 Alinhamento ótimo na matriz do Smith Waterman 8
2.4 Matriz auxiliar E do Gotoh 9
2.5 Matriz auxiliar F do Gotoh 10
2.6 Matriz principal do Gotoh 10
2.7 Alinhamento ótimo na matriz do Gotoh 11
2.8 Pontos de interseção do Myers-Miller [1] 12

3.1 Paralelismo em *wavefront* [2] 14
3.2 *Cells delegation* e *phase division* [2] 16
3.3 Estagios do *MASA-CUDAlign* 17
3.4 Representação geométrica das definições envolvidas na poda de células . . 17
3.5 Representação geométrica das regiões da matriz onde a poda de células é possível [3] 18
3.6 Distribuição da colunas com 4 *GPUs* [2] 19
3.7 Caption 20
3.8 Estapas do *FT-CUDAlign* a cada iteração[4] 21

4.1 Sequência dos arquivos de *checkpoint* de uma *GPU* ao longo do tempo . . 25
4.2 Estrutura do `cell_t` 27
4.3 Diagrama de sequência do *checkpoint* local de linha 28

5.1 Verificação de formato do *checkpoint* salvo 32

Lista de Tabelas

5.1 Sequências Comparadas 30
5.2 Resultados comparação das sequências de 3 *MB* 31
5.3 Resultados comparação das sequências de 7 *MB* 31

Capítulo 1

Introdução

A comparação de sequências biológicas é um problema central em bioinformática, empregada na identificação de mutações, na investigação de relações evolutivas e na análise de similaridades funcionais entre moléculas como *DNA*, *RNA* e proteínas [5]. Para esse fim, algoritmos exatos como Smith–Waterman [6], Gotoh [7] e Myers–Miller [1] calculam o melhor alinhamento possível entre duas sequências por meio de programação dinâmica. Entretanto, esses métodos apresentam custo computacional proporcional ao produto dos comprimentos das sequências, o que torna sua execução direta inviável em cenários de larga escala.

Com o objetivo de superar essa limitação, a ferramenta *MASA-CUDAlign* [8] foi desenvolvida para executar algoritmos de alinhamento de forma eficiente por meio de paralelismo massivo em *GPU*. Nessa abordagem, a matriz de alinhamento é dividida em regiões menores que podem ser processadas simultaneamente, enquanto técnicas como *block pruning* possibilitam descartar partes da matriz que não podem contribuir para o alinhamento ótimo, reduzindo de maneira significativa o volume de computação necessário. À medida que a execução da ferramenta passou a ocorrer em ambientes com múltiplas *GPUs* e por longos períodos, tornou-se necessário incorporar mecanismos adicionais de tolerância a falhas capazes de preservar o progresso realizado ao longo do processamento, e isso deu origem à ferramenta *FT-CUDAlign* [4] , que garante a tolerância a falhas através de *checkpoints* globais específicos de aplicação, que são colunas da matriz de programação dinâmica. Ao ocorrer uma falha, o *FT-CUDAlign* é capaz de re-iniciar a execução a partir do último *checkpoint* global. No entanto, caso as sequências sejam muito grandes, uma grande parte da computação é perdida.

Este trabalho propõe a implementação de um mecanismo complementar a partir do FT-CUDAlign baseado em *checkpoints* locais, no qual cada *GPU* salva periodicamente o estado completo da última linha finalizada em sua região da matriz. Cada uma dessas linhas, denominadas *special rows*, é registrada em um arquivo binário e gerenciada por

um controlador responsável por manter apenas o arquivo mais recente. Essa estratégia evita o acúmulo desnecessário de dados e assegura a existência de um estado válido para eventual recuperação e retomada da execução. Além disso, foi substituída a memória de textura pela memória global, de modo a tornar o FT-CUDAlign compatível com a versão mais recente do CUDAlign.

O restante deste documento está organizado da seguinte maneira. O Capítulo 2 apresenta o problema de comparação de sequências biológicas e os algoritmos exatos para resolvê-lo. No Capítulo 3 são apresentadas as principais características da ferramenta *MASA-CUDAlign*, inclusive da ferramenta *FT-CUDAlign*. O Projecto do mecanismo de *checkpoint* local é apresentado no Capítulo 4 e os resultados experimentais são discutidos no Capítulo 5. Finalmente, o Capítulo 6 conclui o TCC e sugere trabalhos futuros.

Capítulo 2

Comparacao de Sequencias Biologicas

2.1 Visão Geral

As sequências biológicas, como DNA, RNA e proteínas, são moléculas fundamentais que codificam a informação genética e regulam os processos vitais de todos os organismos. O estudo dessas sequências permite compreender desde a organização básica da vida até mecanismos complexos como a expressão gênica, a evolução das espécies e a origem de doenças. Com o avanço das técnicas de sequenciamento de nova geração, tornou-se possível gerar volumes massivos de dados genômicos, sendo necessários métodos computacionais eficientes para analisar, comparar e interpretar essas informações.

A comparação par a par de sequências biológicas consiste em identificar regiões de similaridade entre duas sequências, o que pode revelar homologia evolutiva, conservação funcional ou a presença de elementos genéticos comuns. Esse processo é fundamental para diversas aplicações da bioinformática, como o alinhamento múltiplo de sequências, utilizado para inferir relações evolutivas entre famílias de genes, a construção de árvores filogenéticas, que modelam a história evolutiva de espécies, e o estudo de doenças geneticamente determinadas, incluindo a identificação de marcadores associados ao câncer, mutações patogênicas e variantes genéticas relacionadas a respostas terapêuticas. Além disso, a análise de similaridades desempenha papel central em projetos de anotação genômica, na predição da função de proteínas e no desenho racional de fármacos. [9]

Para realizar essas comparações, algoritmos computacionais são empregados com base em sistemas de pontuações. O objetivo é encontrar o alinhamento ótimo, ou seja, a disposição das sequências que maximize a similaridade. Com base nos resultados, profissionais qualificados conseguem inferir relações evolutivas, identificar elementos conservados, pre-

ver a função de genes desconhecidos e compreender mecanismos moleculares associados a fenótipos de interesse, como doenças ou adaptações ambientais.

2.1.1 Pontuação dos Alinhamentos

O resultado da execução de cada um dos algoritmos mencionados é afetado pelos valores atribuídos às pontuações de *matches*, *mismatches* e *gaps*. A escolha desses parâmetros influencia diretamente a configuração da matriz de alinhamento e, consequentemente, o resultado [10].

Uma das formas de penalizar *gaps* é a pontuação linear, em que cada abertura ou extensão de *gap* tem o mesmo custo fixo, ou seja, a penalidade total por um *gap* de comprimento k é proporcional a k. Embora simples, esse modelo pode não refletir adequadamente a biologia real, pois penaliza da mesma forma um *gap* único e vários *gaps* pequenos separados.

Como alternativa, pode-se utilizar a penalidade por *affine gap* [10], em que se distingue o custo de abertura de um *gap* do custo de sua extensão. Nesse caso, a penalidade total de um *gap* de comprimento k é dada por $open + (k - 1) \times extend$, tornando menos custoso estender um *gap* já existente do que iniciar um novo. Dessa forma, se torna mais próximo do que se observa em sequências biológicas reais, onde inserções ou deleções longas contínuas são mais prováveis do que múltiplos eventos pequenos.

Esse fator pode ser relevante para pesquisadores da área biológica, uma vez que um alinhamento, ainda que computacionalmente ótimo, não é necessariamente o mais provável, fator esse que depende do contexto específico da investigação conduzida. Por exemplo, determinadas aplicações podem atribuir maior importância à ocorrência de *gaps*, enquanto outras priorizam a correspondência exata de pares de bases.

2.1.2 Alinhamento Global e Local

Ao comparar duas sequências biológicas, é possível buscar dois tipos principais de alinhamentos: o alinhamento global, que representa o alinhamento ótimo entre as duas sequências comparadas em sua totalidade, e o alinhamento local, que representa o alinhamento ótimo entre os segmentos mais semelhantes das sequências, conforme exibido na Figura 2.1.

O alinhamento global é encontrado com o algoritmo de Needleman-Wunsch, enquanto o alinhamento local é obtido por meio do algoritmo de Smith-Waterman. Ambos os métodos utilizam programação dinâmica para superar o desafio do tempo ao comparar cadeias muito longas.

Já a avaliação da qualidade do alinhamento baseia-se em uma pontuação probabilística, que expressa a razão entre a chance de o alinhamento ocorrer entre sequências relacionadas e a probabilidade de surgir entre sequências não relacionadas [5].

Sequência 1: A T T C T G C G A T A C G
Sequência 2: C T G G A T T C A A

Alinhamento global:

```
A T T C T G C G A T A C G
- - - | | | - - - - | - - -
- - - C T G G A T T C A A
```

Alinhamento local:

```
G A T
| | |
G A T
```

Figura 2.1: Alinhamento global e local entre duas sequências biológicas.

2.1.3 Programação Dinâmica

A programação dinâmica é uma técnica de otimização amplamente empregada na computação para resolver problemas complexos por meio da decomposição em subproblemas menores, armazenando suas soluções parciais em estruturas de dados auxiliares, como matrizes, a fim de evitar recomputações redundantes [11].

No contexto do alinhamento de sequências, constrói-se uma matriz bidimensional em que uma sequência é disposta ao longo das linhas e a outra ao longo das colunas. Cada elemento da matriz representa a melhor pontuação possível (ou custo mínimo) para alinhar os respectivos prefixos das sequências até aquele ponto. [5].

Ao preencher essa matriz de forma sistemática, utilizando uma equação de recorrência previamente definida, é possível garantir que a solução final seja composta pelas soluções ótimas de todos os subproblemas envolvidos.

Dentre os algoritmos que fazem uso dessa técnica, destacam-se os métodos para alinhamento global e local de sequências, que diferem na forma como a matriz é inicializada, preenchida e percorrida para a reconstrução do alinhamento. Esses aspectos serão explorados nas próximas seções, com foco nos algoritmos de Smith-Waterman e Gotoh.

2.1.4 Algoritmos Exatos

Algoritmos exatos têm como finalidade encontrar soluções ótimas para problemas computacionais, assegurando precisão e completude na resposta final, mesmo que ao custo da

eficiência em tempo ou recursos. Seu principal objetivo é garantir que a solução obtida seja a melhor possível de acordo com os critérios do problema.

De outro lado, existem os algoritmos heurísticos, que priorizam velocidade e buscam por uma solução aproximada, não necessariamente ótima, mas aceitável na busca por um melhor tempo de computação.

Além disso, existem algoritmos exatos que utilizam técnicas heurísticas para aumentar sua eficiência, desde que essas técnicas não afetem o retorno da solução ótima pelo algoritmo. Nesses casos, as heurísticas ajudam a priorizar os melhores caminhos e evitando processamento desnecessário em regiões pouco relevantes para a solução final. O foco do presente trabalho é em algoritmos exatos.

2.2 Algoritmo Smith-Waterman

O algoritmo de Smith-Waterman [6] é um dos principais métodos exatos de comparação de sequências biológicas, sendo amplamente utilizado na bioinformática para o alinhamento local de *DNA*, *RNA* ou proteínas. Sua principal característica é a capacidade de identificar regiões de alta similaridade entre duas sequências, sem exigir que estas sejam alinhadas em sua totalidade.

Neste projeto, foi utilizada uma pontuação de +1 para *match*, -1 para *mismatch* e -2 para *gap*, seguindo o modelo de penalização linear. A complexidade computacional do algoritmo é $O(m \times n)$, tanto em tempo quanto em espaço, onde m e n são os comprimentos das sequências de entradas.

2.2.1 Construção da Matriz de Alinhamento Smith-Waterman

O algoritmo baseia-se na técnica de programação dinâmica para calcular todos os possíveis alinhamentos locais entre duas sequências de entrada, A e B, de comprimentos m e n, respectivamente. Para isso, é construída uma matriz H de dimensões $(m+1) \times (n+1)$, onde cada elemento $H[i][j]$ representa o melhor *score* de alinhamento local entre os prefixos $A[1 \ldots i]$ e $B[1 \ldots j]$, conforme a equação de recorrência 2.1.

Todos os elementos da primeira linha e da primeira coluna da matriz são inicializados com zero, ou seja, $H[0][j] = 0$ e $H[i][0] = 0$, pois não se deseja acumular penalidades fora das regiões de similaridade.

$$[H]H_{i,j} = \max \begin{cases} 0 \\ H_{i-1,j-1} + s(A[i], B[j]) \\ H_{i,j-1} + \text{gap} \\ H_{i-1,j} + \text{gap} \end{cases} \tag{2.1}$$

A função $s(A[i], B[j])$ representa o *score* da substituição entre os símbolos $A[i]$ e $B[j]$, retornando +1 no caso de *match* e -1 em caso de *mismatch*. Já o parâmetro *gap* corresponde à penalidade associada à inserção de um *gap*. A Figura 2.2 apresenta a matriz de exemplo construída com base no algoritmo de Smith-Waterman, considerando duas sequências fictícias.

		A	T	C	T	G
	0	0	0	0	0	0
A	0	1	0	0	0	0
T	0	0	2	0	1	0
G	0	0	0	1	0	2
T	0	0	1	0	2	0
G	0	0	0	0	0	3

Figura 2.2: Matriz do Smith Waterman

2.2.2 Recuperação do Alinhamento Smith Waterman Ótimo (Traceback)

Após preencher a matriz de *scores*, a etapa de recuperação do alinhamento ótimo (conhecida como *traceback*) é iniciada a partir do elemento de maior valor na matriz. A partir desse ponto, o algoritmo segue o caminho que contribuiu para a formação do *score* máximo, prosseguindo no sentido inverso ao do preenchimento da matriz, até encontrar um elemento com valor zero. Esse caminho representa o alinhamento local ótimo, correspondendo ao segmento mais semelhante entre as duas sequências originais. Na Figura 2.3 é apresentada uma visualização do caminho de *traceback* sobre a matriz de pontuação, destacando os elementos que fazem parte do alinhamento ótimo.

		A	T	C	T	G
	0	0	0	0	0	0
A	0	1	0	0	0	0
T	0	0	2	0	1	0
G	0	0	0	1	0	2
T	0	0	1	0	2	0
G	0	0	0	0	0	3

Figura 2.3: Alinhamento ótimo na matriz do Smith Waterman

2.3 Algoritmo Gotoh

O algoritmo de Gotoh [7] é uma extensão dos métodos exatos de alinhamento de sequências, generalizando abordagens como a de Smith-Waterman ao incorporar penalidades distintas para a abertura e a extensão de *gaps*. Essa diferenciação se baseia em observações biológicas, segundo as quais iniciar um *gap* tende a ser um evento mais custoso do que simplesmente estendê-la.

Para este projeto, foi adotada uma pontuação de +1 para *match*, -1 para *mismatch*, -3 para abertura de *gap* e -2 para cada extensão subsequente, conforme o modelo de *affine gap* abordado na Seção 2.1.1. O algoritmo calcula três matrizes dinâmicas para lidar separadamente com os casos de inserção, deleção e alinhamento, o que permite aplicar corretamente a penalidade affine. Ainda assim, a complexidade computacional permanece em $O(m \times n)$ tanto em tempo quanto em espaço, já que as matrizes possuem o mesmo tamanho das sequências de entrada.

2.3.1 Construção da Matriz de Alinhamento Gotoh

A construção do alinhamento no algoritmo de Gotoh também se baseia na técnica de programação dinâmica, mas com uma modelagem mais elaborada para o tratamento de *gaps*. Ao invés de utilizar uma única matriz de pontuação, como no algoritmo de Smith-Waterman, Gotoh introduz três matrizes com dimensões $(m+1) \times (n+1)$: E, F e H, conforme as Equações de recorrência 2.2, 2.3 e 2.4, respectivamente.

A matriz $H[i][j]$ armazena o *score* máximo de um alinhamento que termina com um *match* ou *mismatch* entre os símbolos $A[i]$ e $B[j]$, A matriz $F[i][j]$ representa o melhor *score* de um alinhamento que termina com uma inserção (*gap* na sequência A. Já a matriz $E[i][j]$ representa um alinhamento que termina com uma deleção (*gap* na sequência B).

$$E_{i,j} = \max \begin{cases} H_{i-1,j} - \text{gap_open} \\ E_{i-1,j} - \text{gap_extend} \end{cases} \tag{2.2}$$

$$F_{i,j} = \max \begin{cases} H_{i,j-1} - \text{gap_open} \\ F_{i,j-1} - \text{gap_extend} \end{cases} \tag{2.3}$$

$$H_{i,j} = \max \begin{cases} 0 \\ H_{i-1,j-1} + s(A[i], B[j]) \\ E_{i,j} \\ F_{i,j} \end{cases} \tag{2.4}$$

Assim, a transição para um estado de *gap* envolve uma penalidade de abertura, enquanto a permanência nesse estado ocorre apenas na penalidade de extensão. Essa abordagem garante que alinhamentos longos com *gaps* contínuos não sejam penalizados de forma linear e excessiva, como ocorreria com penalidades fixas.

A inicialização das matrizes depende do tipo de alinhamento desejado. No caso de alinhamento local, todos os elementos das três matrizes podem ser inicialmente zerados: $H[0][j] = F[0][j] = E[0][j] = 0$ e $H[i][0] = F[i][0] = E[i][0] = 0$, permitindo reinicializações quando a pontuação acumulada se torna negativa, de forma análoga ao comportamento do Smith-Waterman.

Nas Figuras 2.4, 2.5 e 2.6 são apresentadas as três matrizes geradas durante a execução do algoritmo.

E		A_1	T_2	C_3	T_4	G_5
		$-\infty$	$-\infty$	$-\infty$	$-\infty$	$-\infty$
A_1	–	-5	-5	-5	-5	-5
T_2	–	-4	-5	-5	-5	-5
G_3	–	-5	-3	-5	-4	-5
T_4	–	-5	-5	-4	-5	-3
G_5	–	-5	-4	-5	-3	-5

Figura 2.4: Matriz auxiliar E do Gotoh

F		A_1	T_2	C_3	T_4	G_5
		$-$	$-$	$-$	$-$	$-$
A_1	$-\infty$	-5	-4	-5	-5	-5
T_2	$-\infty$	-5	-5	-3	-5	-4
G_3	$-\infty$	-5	-5	-5	-4	-5
T_4	$-\infty$	-5	-5	-4	-5	-3
G_5	$-\infty$	-5	-5	-5	-5	-5

Figura 2.5: Matriz auxiliar F do Gotoh

H		A_1	T_2	C_3	T_4	G_5
	0	0	0	0	0	0
A_1	0	1	0	0	0	0
T_2	0	0	2	0	1	0
G_3	0	0	0	1	0	2
T_4	0	0	1	0	2	0
G_5	0	0	0	0	0	3

Figura 2.6: Matriz principal do Gotoh

2.3.2 Recuperação do Alinhamento Ótimo (*Traceback*) Gotoh

Após o preenchimento completo das matrizes H, E e F, o alinhamento ótimo é reconstruído por meio do processo de *traceback*. No caso de alinhamento local, esse procedimento tem início a partir do elemento com o maior valor dentre todas as matrizes, pois esse valor corresponde ao *score* máximo atingido por qualquer subalinhamento entre as duas sequências. A partir desse ponto, percorre-se o caminho que originou o valor máximo, levando em conta a origem da pontuação (se veio de um *match/mismatch*, abertura ou extensão de *gap*), de forma semelhante ao que ocorre no algoritmo de Smith-Waterman.

Durante esse percurso, o algoritmo consulta qual das matrizes forneceu a pontuação atual, e retrocede de acordo com a equação de recorrência correspondente. A trajetória é seguida até que um valor zero seja alcançado, indicando o início do alinhamento local ótimo. Assim, o caminho percorrido no *traceback* define as posições que compõem o alinhamento final entre os segmentos de maior similaridade nas duas sequências originais.

H		A_1	T_2	C_3	T_4	G_5
	0	0	0	0	0	0
A_1	0	1	0	0	0	0
T_2	0	0	2	0	1	0
G_3	0	0	0	1	0	2
T_4	0	0	1	0	2	0
G_5	0	0	0	0	0	3

Figura 2.7: Alinhamento ótimo na matriz do Gotoh

A Figura 2.7 modela o caminho de *traceback* sobre a matriz de pontuação, destacando os elementos que fazem parte do alinhamento ótimo.

2.4 Algoritmo Myers-Miller

Para contornar o problema da complexidade de memória que a programação dinâmica apresentada no Smith-Waterman e no Gotoh traz, Hirschberg [12] desenvolveu um novo método que emprega a técnica de dividir para conquistar, possibilitando o cálculo do *score* ótimo sem a necessidade de armazenar toda a matriz de programação dinâmica, mantendo apenas as linhas atual e anterior.

A partir dessa aborgadem, o algoritmo de Myers-Miller [1] propõe uma implementação do método de Hischberg que realiza o *traceback* em memória linear.

O algoritmo de Myers-Millers é, portanto, uma implementação em espaço linear do algoritmo de Gotoh, a partir do método proposto por Hischberg para comparação de alinhamento de sequencias.

2.4.1 Construção da Matriz de Alinhamento Myers-Miller

No lugar de armazenar integralmente as matrizes H, F e E, a estratégia consiste em realizar o cálculo dos *scores* de alinhamento utilizando apenas os valores da linha atual e da linha anterior, suficientes para determinar a direção do alinhamento ótimo.

O processo inicia-se com duas passagens principais: uma em sentido direto e a outra no sentido contrário. A interseção entre essas passagens identifica o ponto médio do alinhamento ótimo, permitindo a divisão do problema em duas partes menores e independentes.

Assim, a recorrência da computação dos *scores*, adaptada à estrutura do algoritmo, pode ser expressa conforme as Equações 2.5, 2.6 e 2.7, onde $s(A[i], B[j])$ representa a função de similaridade entre os símbolos $A[i]$ e $B[j]$, enquanto gap_open e gap_extend denotam as penalidades de abertura e extensão de *gaps*, respectivamente.

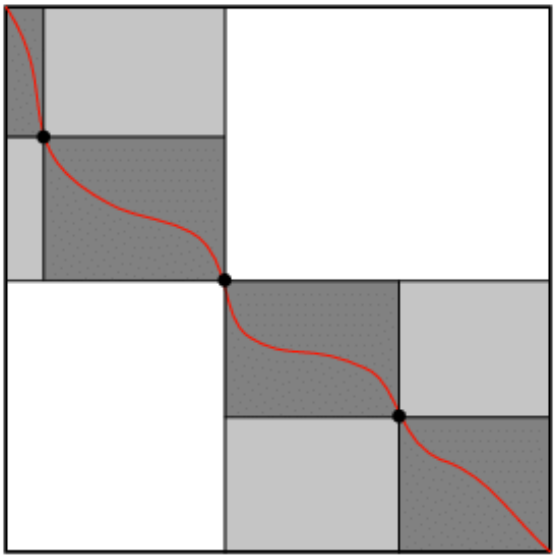


Figura 2.8: Pontos de interseção do Myers-Miller [1]

$$E_{i,j} = \max \begin{cases} H_{i-1,j} - \text{gap_open} \\ E_{i-1,j} - \text{gap_extend} \end{cases} \tag{2.5}$$

$$F_{i,j} = \max \begin{cases} H_{i,j-1} - \text{gap_open} \\ F_{i,j-1} - \text{gap_extend} \end{cases} \tag{2.6}$$

$$H_{i,j} = \max \begin{cases} H_{i-1,j-1} + s(A[i], B[j]) \\ E_{i,j} \\ F_{i,j} \end{cases} \tag{2.7}$$

Após encontrar o ponto médio, o algoritmo aplica-se recursivamente às subinstâncias, até que estas sejam suficientemente pequenas para serem resolvidas diretamente, de acordo com a Figura 2.8.

2.4.2 Recuperação do Alinhamento Ótimo (*Traceback*) Myers-Miller

O alinhamento ótimo completo é então reconstruído a partir da concatenação dos alinhamentos parciais obtidos em cada nível da recursão.

Dessa forma, o algoritmo Myers-Miller consegue reduzir significativamente a complexidade espacial da construção do alinhamento, de $O(mn)$ para $O(n)$, mantendo a mesma

complexidade temporal $O(mn)$ do algoritmo de Gotoh. Essa característica torna o método especialmente útil para o alinhamento de sequências biológicas longas.

Capítulo 3

Evolução da Ferramenta *MASA-CUDAlign*

3.1 *MASA-CUDAlign*

O *MASA-CUDAlign* é uma ferramenta de alinhamento de sequências biológicas que se destaca pela capacidade de alinhar sequências de *DNA* de grande porte de forma exata, ou seja, retornando sempre o alinhamento ótimo.

O processo de alinhamento no *CUDAlign* [13] é dividido em seis estágios conforme a Figura 3.3, sendo o primeiro estágio o mais intensivo em termos computacionais. Nesse estágio, é realizada a primeira fase do algoritmo de Smith-Waterman (Seção 2.2) com penalidade por *affine gap*, o qual tem como objetivo calcular a matriz completa e identificar o *score* ótimo e sua posição.

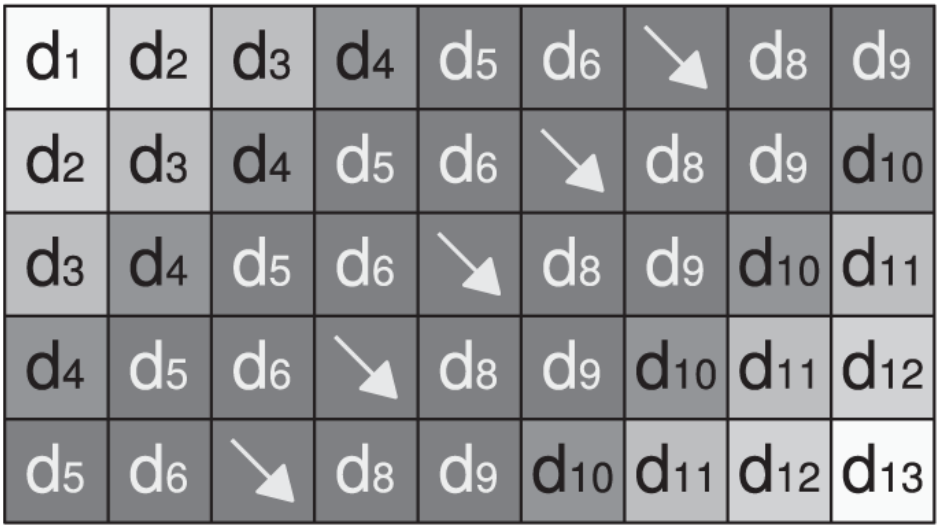


Figura 3.1: Paralelismo em *wavefront* [2]

Para viabilizar o alinhamento de sequências longas em tempo viável, o *CUDAlign* utiliza o paralelismo massivo oferecido pela arquitetura *CUDA*, através da técnica conhecida como *wavefront*, observada na Figura 3.1. Essa técnica organiza a matriz em antidiagonais de blocos (paralelismo externo), permitindo o processamento paralelo de múltiplos blocos sem violar as dependências de dados. Dentro de cada bloco, os elementos são também organizados em antidiagonais (paralelismo interno), o que maximiza o aproveitamento das *threads* da *GPU*.

Na prática, cada bloco *CUDA* é responsável por processar (αT) linhas da matriz, onde α representa a quantidade de linhas por *thread*, e T o número de *threads* por bloco. Como os blocos de uma mesma antidiagonal não têm dependências entre si, podem ser executados em paralelo. Após a conclusão de todos os blocos de uma antidiagonal, a próxima pode ser processada, garantindo tanto a correção dos resultados quanto a eficiência computacional.

Para reduzir ainda mais o tempo de execução do Estágio 1, o *CUDAlign* adota duas otimizações: *Cells Delegation* e *Phase Division*.

A técnica de *Cells Delegation* busca manter o paralelismo contínuo entre os blocos *CUDA* ao longo das antidiagonais externas, evitando o esvaziamento e preenchimento completo da *wavefront* a cada nova antidiagonal. Para isso, os blocos passam a ter formato de paralelogramo, em vez do formato retangular convencional, visto na Figura 3.2, permitindo que blocos adjacentes deleguem células entre si, o que contribui para um uso mais eficiente da *GPU* durante quase toda a execução do estágio.

Já a técnica de *Phase Division* tem como objetivo evitar condições de corrida (*hazards*) durante o processamento de cada antidiagonal externa, exibida na Figura 3.2. Para isso, cada bloco é processado em duas fases: a *short phase*, que calcula as primeiras antidiagonais internas, e a *long phase* que processa as antidiagonais restantes. Cada fase é executada por um *kernel* distinto, sendo que o *kernel* da *long phase* é otimizado para maior desempenho.

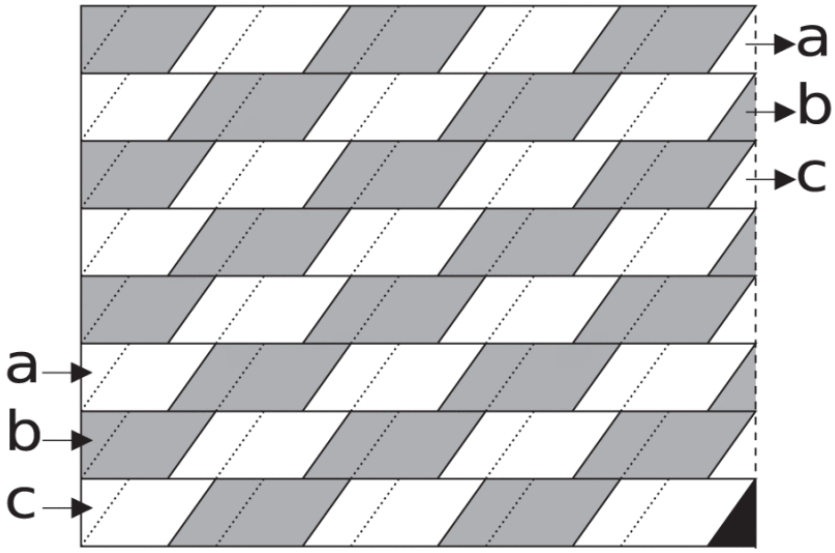


Figura 3.2: *Cells delegation* e *phase division* [2]

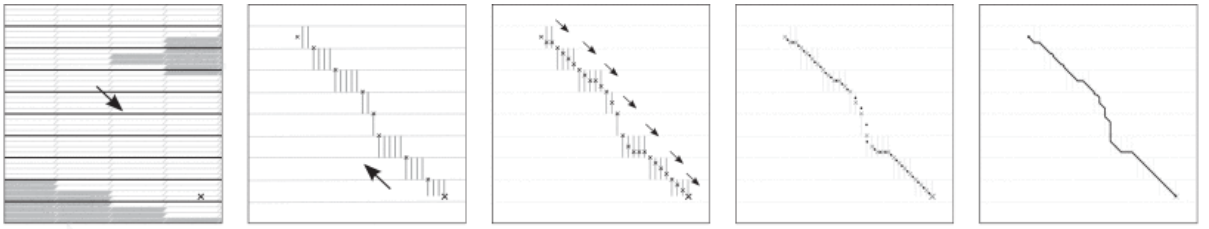
No Estágio 2, são recuperados os *crosspoints*, isto é pontos estratégicos previamente armazenados durante a fase de cálculo. Esses pontos indicam posições relevantes na matriz que pertencem ao caminho do alinhamento ótimo. A partir deles, é possível delimitar sub-regiões independentes entre pares consecutivos de *crosspoints*.

O Estágio 3 atua sobre essas sub-regiões, processando-as de maneira paralela. Para isso, utiliza-se a mesma abordagem do Estágio 2, porém aplicada a regiões menores, aproveitando a independência entre elas para acelerar a recuperação de mais pontos do alinhamento ótimo.

O Estágio 4 aplica o algoritmo de Myers-Miller (Seção 2.4) de forma iterativa dentro dessas regiões delimitadas. Seu objetivo consiste em continuar refinando o alinhamento, dividindo as regiões até que cada partição atinja um tamanho suficientemente pequeno.

O Estágio 5 realiza a reconstrução detalhada do alinhamento nas partições menores utilizando o algoritmo de Needleman-Wunsch, o qual é uma variação global do Smith-Waterman, preenchendo toda a matriz de similaridade para encontrar o melhor alinhamento. Assim como Gotoh (Equação 2.3.1), ele pode utilizar penalidades por abertura e extensão de gaps, garantindo que a reconstrução do alinhamento seja feita com precisão.

Caso for necessário, pode-se também ter o Estágio 6 que encerra o processo oferecendo uma visualização textual do alinhamento final, permitindo a análise e interpretação do resultado.



(a) Stage 1 finds the optimal score and its position. Special rows are stored on disk.

(b) Stage 2 finds crosspoints between optimal alignment and special rows.

(c) Stage 3 finds more crosspoints over special rows stored from previous stages.

(d) Stage 4 executes Myers and Miller's algorithm between successive crosspoints.

(e) Stage 5 obtains the complete alignment between each successive crosspoints.

Figura 3.3: Estagios do *MASA-CUDAlign*

3.1.1 Block Pruning

A técnica de poda de blocos (*Block Pruning*) foi proposta no *CUDAlign 2.1* [8] e é utilizada no Estágio 1 do alinhamento com o objetivo de reduzir o número de operações necessárias no preenchimento da matriz. Em vez de computar toda a matriz, a técnica identifica blocos que podem ser descartados por não terem chance de contribuir para o alinhamento ótimo.

A identificação dessas células podáveis ocorre a partir da análise de suas coordenadas e do *score* atual. Para uma célula $H(i, j)$, calcula-se a distância até o fim da matriz tanto em linhas (Δi) quanto em colunas (Δj). Com isso, o maior *score* que cada célula pode receber é dado por: $Hmax(i, j) = H(i, j) + min(\Delta i, \Delta j) \times match$, mostrado na Figura 3.4.

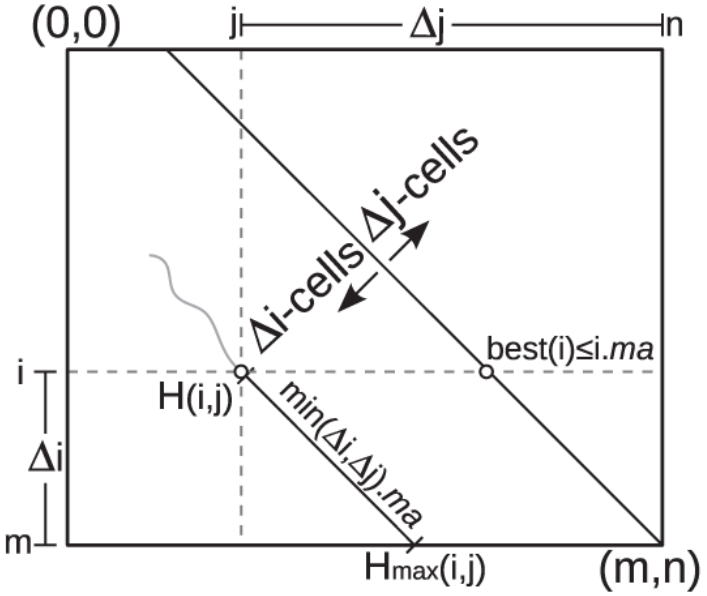


Figura 3.4: Representação geométrica das definições envolvidas na poda de células

Se esse valor for menor que o melhor *score* global já encontrado até a linha i ($best(i)$), então a célula (i, j) pode ser descartada, reduzindo significativamente a área da matriz que precisa ser processada, aumentando o desempenho da aplicação, como é possível observar na Figura 3.5 onde regiões sombreadas indicam áreas que as células podem ser descartadas.

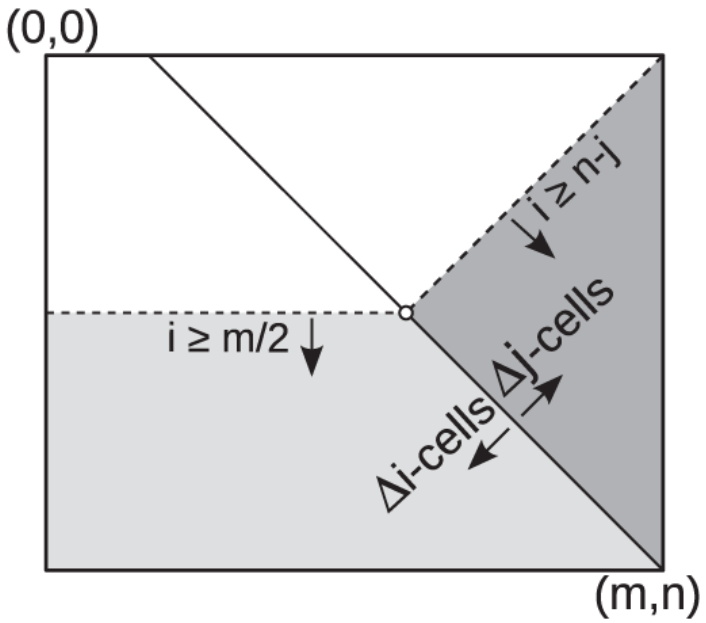


Figura 3.5: Representação geométrica das regiões da matriz onde a poda de células é possível [3]

3.1.2 *CUDAlign Multi-GPU*

A fim de reduzir ainda mais o tempo de execução do *MASA-CUDAlign*, foi implementado um módulo de execução em Múltiplas *GPUs*, dando origem ao *CUDAlign* 3.0 [14]. Essa abordagem baseia-se na subdivisão da matriz de alinhamento em um conjunto de colunas, delegando a cada *GPU* a responsabilidade por processar um subconjunto de colunas. O tamanho das colunas pode variar, de forma a equilibrar a carga de trabalho com a capacidade computacional de cada *GPU*, especialmente em ambientes heterogêneos, conforme a Figura 3.6. Essa estratégia visa otimizar o tempo de processamento e maximizar o número de linhas processadas por segundo. A última célula de cada linha pertence à coluna de borda, que deve ser transmitida para a próxima *GPU*, a realização dessa tarefa com eficiência é fundamental para maximizar o tempo de atividade das *GPUs*.

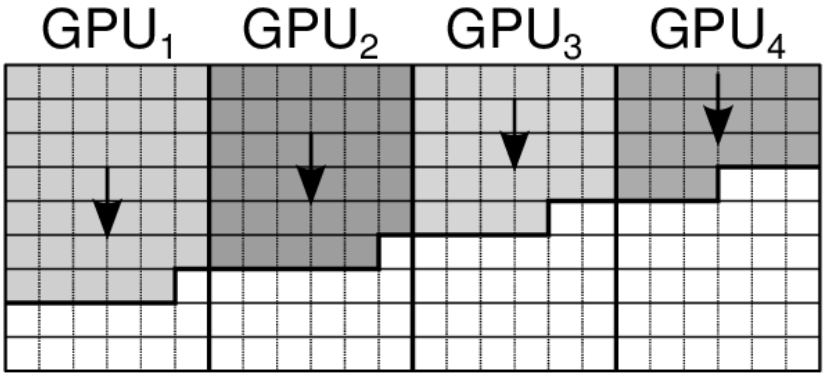


Figura 3.6: Distribuição da colunas com 4 *GPUs* [2]

Cada *GPU* possui três *threads* associadas, uma responsável por controlar a execução e outras duas responsáveis pela comunicação entre processos. Assim, toda *GPU* estabelece comunicação com as *GPUs* adjacentes, recebendo dados da *GPU* à esquerda por meio de seu *input buffer* e enviando dados para a *GPU* à direita por meio de seu *output buffer*. Dessa forma, forma-se uma cadeia de *GPUs*, em que cada unidade depende diretamente da anterior e da seguinte para garantir a continuidade da execução.

3.2 *DynamicBP*

O *DynamicBP* [15] é a versão do *MASA-CUDAlign* com suporte a *block pruning* e múltiplas *GPUs*. Seu funcionamento é baseado no processamento da matriz de alinhamento em ciclos sucessivos, nos quais apenas uma parte da matriz é computada por vez. O número total de ciclos é determinado por $breakpoints + 1$, onde cada *breakpoint* indica um ponto de divisão da matriz.

Ao término de cada ciclo, o sistema coleta métricas (linhas por segundo) utilizadas para avaliar se a distribuição de carga entre as *GPUs* está balanceada. Caso seja identificado algum desequilíbrio, uma nova redistribuição é realizada antes do início do ciclo seguinte, adaptando-se dinamicamente ao comportamento observado durante a execução.

Durante o processamento, cada *GPU* troca uma métrica de melhor *score* com as demais utilizando uma topologia em anel. Isso significa que cada *GPU* envia sua métrica para a próxima na cadeia, e a última envia de volta para a primeira. Essa troca é feita de forma assíncrona e permite que todas as *GPUs* utilizem um valor de *score* global atualizado, o que melhora a efetividade da poda nas regiões recebidas por cada *GPU*.

Além disso, por conta das dependências de dados horizontais na matriz, ao final de cada iteração, cada *GPU* deve enviar as células da última coluna de sua região (as células de borda) para a *GPU* seguinte. A última *GPU* do grupo armazena sua coluna final

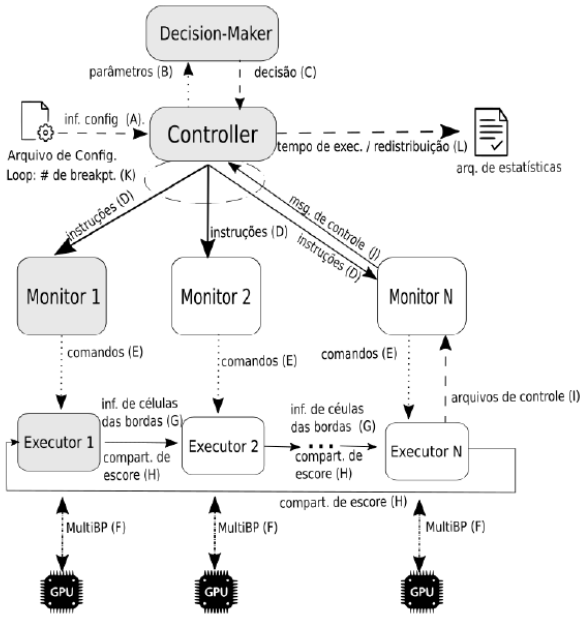


Figura 3.7: Caption

em um arquivo, que será lido pela primeira *GPU* na iteração seguinte. Esse mecanismo assegura a continuidade da computação sem perda de informações nas bordas das regiões processadas.

Em suma, a execução do *DynamicBP* é formada por quatro módulos principais: o *Executor*, responsável por executar o *MASA-CUDAlign* em cada *GPU*; o *Monitor*, que observa e sinaliza eventos importantes da execução; o *Controller*, que gerencia a redistribuição de carga e emite comandos; e o *Decision Maker*, que decide quando e como ajustar a execução com base nas métricas coletadas, conforme mostra a Figura 3.7. Com isso, esses módulos garantem uma execução eficiente, mesmo em ambientes com múltiplas *GPUs* e características heterogêneas.

3.3 *FT-CUDAlign*

O *FT-CUDAlign* [4] foi construído a partir da versão *Dynamic-MultiBP* do *MASA-CUDAlign* 3.2. Nesse contexto, tanto o *MASA-CUDAlign* quanto o *Controller* podem identificar falhas, mas somente o *Controller* é capaz de realizar o tratamento. Assim, caso uma falha seja identificada pelo *MASA-CUDAlign*, ele a reportará para o módulo do *Controller*.

Cada iteração possui 3 momentos principais referentes ao módulo de recuperação de falhas, exemplificados na Figura 3.8.

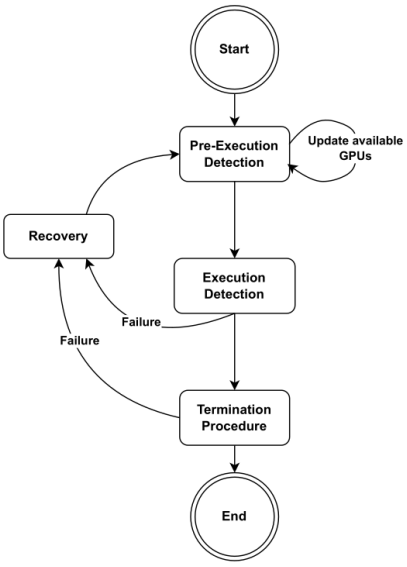


Figura 3.8: Estapas do *FT-CUDAlign* a cada iteração[4]

A primeira etapa é a Detecção Pré-Execução, em que o *Controller* testa a conexão com as *GPUs* disponíveis. Caso a conexão com alguma *GPU* não seja bem sucedida, ela será removida da lista de máquinas disponíveis e a execução ocorrerá apenas com as *GPUs* restantes, desde que haja ao menos duas.

Em seguida, vem a Detecção Durante a Execução, que verifica se ocorreu alguma falha durante o processamento da matriz de programação dinâmica. Caso aconteça alguma falha durante esse período, a execução é interrompida, a conexão com cada máquina é reavaliada e a lista de *GPUs* ativas é novamente atualizada, e a conexão continua a partir do último *checkpoint*.

Por fim, para terminar o processamento de uma matriz de programação dinâmica o programa executa o Procedimento de Término, após o qual será iniciada a próxima iteração.

3.3.1 Detecção Pré-Execução

Antes de iniciar a execução das matrizes, o *FT-CUDAlign* verifica quais *GPUs* estão funcionando. Nesse momento, ele lista os endereços *IP* de todas *GPUs* do sistema e tenta estabelecer conexão com cada uma delas. Para tanto, o *Controller* segue uma política de *retry* em que ele tenta estabelecer a conexão por um número determinado de vezes, caso em alguma dessas vezes a conexão seja bem estabelecida, o *Controller* segue a execução. Todas *GPUs* em que o *Controller* não conseguiu se conectar após todos os retries definidos são retiradas da lista de *GPUs* disponíveis. Após isso, o *Controller* reenvia comandos aos *Balancers* das *GPUs* ativas para reequilibrar a distribuição da execução.

3.3.2 Detecção Durante a Execução

Os módulos Monitor inicializam a execução do *CUDAlign*. Estes estabelecem comunicação entre si de modo que cada *GPU* se comunique com a *GPU* que a precede e com a *GPU* logo após ela. Dessa forma, uma máquina espera por dados de sua predecessora e envia dados para sua sucessora. O módulo de tolerância a falhas verifica os retornos das funções de comunicação (*sockets*) entre as máquinas para verificar caso alguma falha de fato ocorra.

Esse mesmo mecanismo é utilizado também para a detecção de falhas, de modo que apenas as *GPUs* envolvidas na comunicação são capazes de identificar que uma falha ocorreu, isto é, apenas as máquinas adjacentes.

A última *GPU* salva em disco a última coluna da iteração, que coincide com o *break-point*. Essa coluna terá dois propósitos, o primeiro é dar continuidade ao processamento da matriz na próxima iteração, o segundo é servir de *checkpoint* global para o mecanismo de recuperação de falhas. Quando uma falha ocorre, esse mecanismo inicia a recuperação a partir da última coluna salva em disco, descartando todo o processamento posterior a esse *checkpoint* e reiniciando a execução a partir dele.

No momento de recuperação, o *Controller* refaz o procedimento que verifica quais *GPUs* estão funcionando, tentando estabelecer conexão com cada uma delas. Nesse momento, *GPUs* podem falhar, fazendo com que a próxima iteração ocorra com menos *GPUs*.

3.3.3 Recuperação

O Processo de recuperação de falhas é gerenciado pelo módulo do *Controller*. Quando uma falha ocorre em uma dada *GPU*, ela é detectada por uma das *GPUs* adjacentes àquela que falhou, nesse momento o *Controller* inicia o processo de recuperação. Inicialmente, ele coloca as demais máquinas em *standby* e mata todos os processos do *CUDAlign*. Com todas as *GPUs* esperando, ele tenta reestabelecer a conexão com cada máquina, repetindo o processo de *retry* e, com isso, atualizando a lista de *GPUs* disponíveis. Com a nova lista de *GPUs*, o *Controller* busca pelo último *checkpoint* e prepara a próxima iteração a partir dele.

3.3.4 Protocolos de Término

Sincronização da Destruição do *Socket*

Um outro momento em que o processo de recuperação de falhas deve atuar é em garantir que o processamento da matriz finalize adequadamente, isso é, verificar falhas em todas as etapas de encerramento para conseguir recuperar falhas nos momentos finais.

Para isso, o *FT-CUDAlign* deve verificar ainda os procedimentos de escrita do arquivo de estatísticas, terminar de salvar o *checkpoint* em disco e destruir estruturas de dados (*buffer*, *socket*, etc).

Como a detecção de uma falha é feita por uma *GPU* adjacente a que falhou e cada *GPU* pode terminar seu processamento em um momento diferente, caso a penultima *GPU* termine seu processo antes da última, ela não poderá mais identificar falhas na execução da última *GPU*. Para solucionar esse problema, uma *GPU* só pode destruir seu *socket* de escrita quando a *GPU* posterior a ela enviar uma mensagem informando que terminou de executar, permitindo, assim, que o *socket* de escrita seja destruído e o processo finalizado.

Sincronização do Término de Execução

Ao final da execução, a última *GPU* escreve dois arquivos iguais, de modo que ambos informam ao *Controller* que a iteração corrente foi finalizada e que ele já pode enviar os comandos da próxima iteração. O primeiro arquivo é escrito antes da destruição do *socket*, e o segundo após essa destruição. Dessa forma, quando o Controller recebe o primeiro arquivo, ele espera pela criação do segundo arquivo durante um certo tempo, ao final do qual ocorre um *timeout*. Se o segundo arquivo não for criado antes do *timeout*, o *Controller* assume que ocorreu uma falha durante o processo de destruição do *socket*, assim iniciando o protolo de recuperação. Dessa forma, o *FT-CUDAlign* garante que todos os processos do *CUDAlign* tiveram sucesso antes de finalizar a execução.

Capítulo 4

Projeto do *Checkpoint* Local para o *FT-CUDAlign*

4.1 Visão geral

O mecanismo de tolerância a falhas originalmente implementado no *FT-CUDAlign* baseia-se exclusivamente em *checkpoints* globais gerados em pontos específicos da execução, chamados de *breakpoints*. Nessas etapas, o módulo *Controller* coordena todas as *GPUs* para que enviem seus dados de fronteira, produzindo um estado global consistente capaz de ser restaurado após uma falha. Embora esse método seja funcional, ele apresenta uma limitação importante, pois todo o progresso realizado entre dois *breakpoints* é perdido caso ocorra uma interrupção inesperada.

Para mitigar esse problema, este trabalho propõe um novo mecanismo complementar baseado em *checkpoints* locais, produzidos de forma independente por cada *GPU*. A ideia central consiste em registrar periodicamente a última linha processada por cada instância do *CUDAlign*, permitindo que, após uma falha, a execução seja retomada o mais próximo possível do ponto de interrupção, sem depender de sincronizações globais. Essa estratégia é viável porque cada *GPU* opera sobre um subconjunto de colunas específico da matriz de alinhamento e interage com as demais apenas por meio da troca de colunas de borda. Como o cálculo dentro de cada subconjunto de colunas é determinístico e avança linha a linha, conhecer o estado da última linha finalizada é suficiente para retomar corretamente a computação.

A cada conjunto de linhas processadas, a *GPU* salva seu estado em um arquivo temporário, posteriormente convertido em *checkpoint* válido por meio de uma operação de renomeação atômica. Como observado na Figura 4.1, no máximo dois arquivos são mantidos simultaneamente, e *checkpoints* antigos são removidos para economizar espaço e evitar acúmulo desnecessário. A remoção do arquivo mais antigo é feita apenas depois

que um novo *checkpoint* terminar de ser escrito, isso garante que caso ocorra uma falha durante a escrita, a renomeação ou a remoção do arquivo, o *checkpoint* anterior permanece preservado, garantindo um ponto seguro de recuperação. O conteúdo de cada linha é armazenado em formato binário contendo toda a estrutura necessária para restaurar o cálculo.

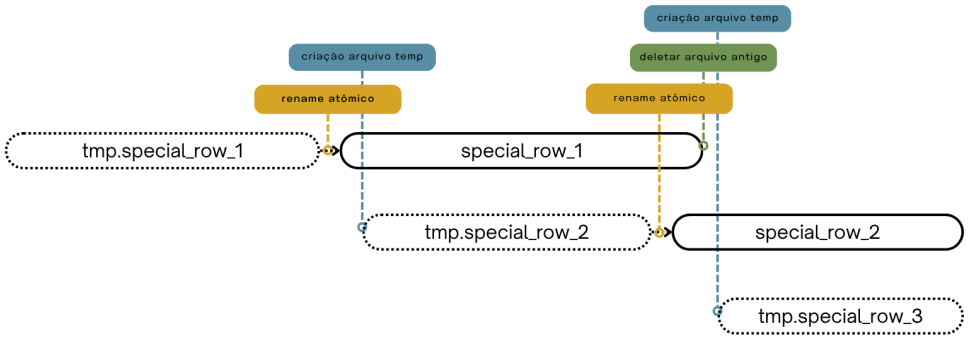


Figura 4.1: Sequência dos arquivos de *checkpoint* de uma *GPU* ao longo do tempo

Toda essa funcionalidade foi integrada ao *FT-CUDAlign* de forma modular, por meio das classes `SpecialRowsPartition` e `SpecialRowWriter`, sem alterar o fluxo principal do sistema nem interferir no mecanismo global existente. Assim, o *checkpoint* local constitui uma camada adicional de segurança, reduz o retrabalho após falhas e estabelece a base para um mecanismo completo de recuperação no nível de linha.

É importante destacar que o fluxo que aciona o `SpecialRowsPartition`, e consequentemente a chamada ao `SpecialRowWriter`, é atualmente executado apenas quando o *CUDAlign* é iniciado com uma única *GPU* e com a *flag* `-disk-size`. Nessa configuração, o processamento percorre o caminho responsável pela geração das `SpecialRows`. Em contraste, ao executar o *FT-CUDAlign* com duas ou mais de *GPUs*, o que implica na execução via *Balancer/Controller*, esse fluxo não é acionado e, portanto, os *checkpoints* locais ainda não são produzidos nesse modo. Essa limitação corresponde ao estágio atual da integração e será endereçada em trabalhos futuros.

Além do mecanismo de *checkpoint* local, este trabalho integra ao *FT-CUDAlign* a atualização estrutural apresentada em [16], que substitui o uso de memória de textura por memória global para armazenar as sequências de entrada na *GPU*. O trabalho referenciado demonstra que, embora a memória de textura tenha sido utilizada em versões anteriores do *CUDAlign* para reduzir a latência de acesso em padrões de leitura sequencial, sua *API* foi descontinuada a partir do *CUDA* 10.1, impedindo a compilação em versões recentes da plataforma. A substituição por memória global eliminou essa dependência e removeu o limite artificial de 134 *MB* que restringia o processamento de sequências longas. Conforme evidenciado em [16], essa modificação não apenas preserva a eficiência do alinhamento para sequências menores, como também melhora o desempenho para sequências acima do limite anterior, devido à eliminação do particionamento artificial. Dessa forma, o *FT-CUDAlign* atualizado mantém compatibilidade com as versões atuais do *CUDA* e processa sequências de tamanho arbitrário limitadas apenas pela memória global da *GPU*.

4.2 Tomada de *Checkpoint* local

A criação de um *checkpoint* local ocorre de maneira periódica e independente em cada instância do *CUDAlign*. Além das colunas de borda tradicionalmente utilizadas para continuidade entre iterações, cada *GPU* começa a salvar um conjunto selecionado de linhas do seu bloco de processamento, denominadas `SpecialRows`. Cada `SpecialRow` corresponde a uma linha da matriz de alinhamento local em uma dada *GPU* e contém todas as informações para retomar o processamento após uma falha.

Para organizar esses dados, cada processo do *CUDAlign* mantém um diretório próprio de *checkpoints*, no qual cada arquivo corresponde exatamente a uma `SpecialRow`. A escrita é realizada em formato binário, garantindo compactação e rapidez no armazenamento. Após ser gravado, o arquivo é lido novamente para desserialização, permitindo restaurar a estrutura original da linha. Essa verificação imediata assegura que o conteúdo foi salvo corretamente e evita a criação de *checkpoints* inválidos que poderiam comprometer uma recuperação futura.

A estrutura de cada célula armazenada em uma `SpecialRow` é apresentada na Figura 4.2. Cada célula contém o valor de H, além de um campo unificado para E ou F, refletindo o fato de que essas componentes compartilham a mesma região de memória no modelo de cálculo empregado pelo *CUDAlign*.

Assim, uma `SpecialRow` consiste em um vetor de estruturas `cell_t`, representando o estado da linha no momento de sua finalização. Esse mecanismo permite reconstruir o estado interno de uma *GPU* com granularidade de linha, possibilitando a retomada da execução a partir do último *checkpoint* local válido.

```
typedef struct {
 int h;
 union {
  int f;
  int e;
 };
} __attribute__ ((aligned (8))) cell_t;
```

Figura 4.2: Estrutura do `cell_t`

4.3 Projeto do *checkpoint* local

Como cada processo do *CUDAlign* opera sobre uma fatia da matriz, os *checkpoints* não precisam ser coordenados entre os processos em execução. Isto garante que cada bloco (processado em uma *GPU*) tenha seu próprio *checkpoint* de linha, permitindo que, em caso de falha, um nova execução seja iniciada a partir da última linha salva naquela *GPU*.

O sistema de *checkpoints* é implementado a partir de duas classes, a `SpecialRowsPartition`, que atua como um controlador para o sistema de salvamento de linhas, e a `SpecialRowWriter`, que é instanciada dentro da `SpecialRowsPartition` e realiza as operações de salvar e apagar arquivos.

O processo de escrita de um arquivo é realizado pelo método `write` do `SpecialRowWriter`. Caso o arquivo ainda não exista, ele cria um arquivo temporário com o prefixo *tmp.*, no qual a escrita ocorrerá. Esse método recebe em binário os dados da struct de uma célula de uma linha e escreve no arquivo temporário correspondente. Quando a linha é concluída, ele chama uma operação de *rename* atômica que remove o prefixo do nome do arquivo.

Já o método `delete` apaga todos os arquivos exceto o arquivo mais recente. Na prática, é garantido que existam no máximo dois arquivos ao mesmo tempo, assim o método vai sempre deletar o arquivo mais antigo. Vale ressaltar que esse método avalia apenas arquivos completos, ignorando temporários, de modo que um arquivo não será deletado durante o processo de escrita.

A seleção do arquivo mais antigo é feita de acordo com o tempo da última escrita de cada um. Como cada instância escreve apenas um arquivo por vez e depois de concluído ele não é mais atualizado, é garantido que o arquivo com tempo de escrita mais recente será sempre referente a uma linha de índice maior.

Nesse cenário, a ordem das operações é de grande importância, pois ela garante a integridade dos *checkpoints* em caso de falha durante a escrita de uma linha. Isso é

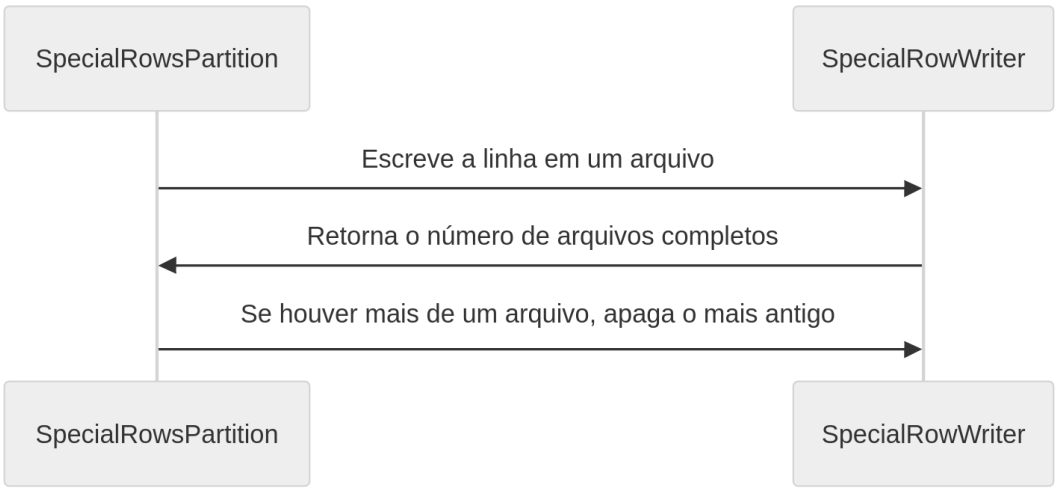


Figura 4.3: Diagrama de sequência do *checkpoint* local de linha

possível por o sistema só apagar um *checkpoint* antigo se já houver outro mais recente concluído, de modo que sempre exista ao menos um *checkpoint* completo.

Dessa forma, para salvar um *checkpoint* local, cada instância executa os passos a seguir (Fig. 4.3):

1. `SpecialRowsPartition` chama o método `write` de `SpecialRowWriter` para persistir a última linha em um novo arquivo;
2. `SpecialRowWriter` retorna a quantidade de arquivos na pasta de *checkpoints* do *CUDAlign*;
3. se existirem pelo menos dois *checkpoints*, o `SpecialRowWriter` chama o método `delete` e remove o mais antigo, mantendo apenas o mais recente.

Como cada *GPU* processa uma área independente da matriz, compartilhando apenas o conteúdo da coluna de borda, verifica-se que no caso de uma *GPU* falhar, as máquinas anteriores a ela continuarão a execução até que seu *buffer* fique cheio, caso em que elas começam a esperar pelo esvaziamento do *buffer* pela *GPU* em recuperação, e as posteriores param esperando pelas mensagens da máquina em falta. Nesse cenário, será possível que a *GPU* em recuperação, por sua vez, após recuperar o estado do último *checkpoint* local, retome o processamento e volte a enviar e consumir informações, desencadeando assim o

retorno do processamento do *FT-CUDAlign*. Por meio dessa abordagem, os *checkpoints* locais nessa implementação não correm o risco de sofrer de um efeito cascata, uma vez que eles serão utilizados de modo isolado para a recuperação do bloco selecionado.

4.4 Memória de textura

O desempenho do *FT-CUDAlign* depende diretamente da eficiência com que as *GPUs* acessam as sequências de entrada ao longo do cálculo da matriz de alinhamento. Cada linha processada exige leituras constantes e previsíveis sobre ambas as sequências, com padrões intensivos em acesso e essencialmente determinísticos. Na implementação original do *MASA-CUDAlign*, essas sequências eram armazenadas em memória de textura, que possuía um *cache* dedicado otimizado para padrões de leitura sequencial.

No entanto, conforme demonstrado no artigo [16], a *API* de manipulação de textura (*Texture Reference Management API*) foi descontinuada a partir do *CUDA* 10.1, tornando o código incompatível com versões recentes da plataforma. Além disso, a memória de textura unidimensional impunha um limite rígido de 134 *MB* (aproximadamente 134 milhões de pares de bases), o que forçava o particionamento artificial de sequências longas prejudicando o desempenho.

Este trabalho também integra ao *FT-CUDAlign* a atualização proposta no artigo [16], que substitui completamente o uso de memória de textura por memória global. A implementação envolveu três etapas principais:

1. `Remoção das texturas do barramento horizontal:` Todas as chamadas à função `tex1Dfetch` foram substituídas por acessos diretos a ponteiros na memória global, eliminando a dependência da *API* de textura;
2. `Substituição das texturas das sequências de entrada:` As variáveis de textura que armazenavam as sequências de *DNA* foram substituídas por ponteiros para memória global. Foram criadas as funções `setGlobalSequences` e `unsetGlobalSequences` para gerenciar esses ponteiros na *GPU*;
3. `Eliminação do limite de tamanho:` A constante que restringia as sequências a 134 *MB* foi removida, permitindo que o *FT-CUDAlign* processe sequências de tamanho arbitrário, limitadas apenas pela memória disponível na *GPU*.

Os resultados apresentados no artigo [16] mostram que essa substituição mantém compatibilidade com versões recentes do *CUDA*, preserva a eficiência para sequências menores e melhora o desempenho em até 16,6% para sequências acima de 134 MBP, devido à eliminação do overhead de particionamento.

Capítulo 5

Resultados Experimentais

5.1 Ambiente de Testes

Os experimentos foram realizados no Laboratório de Sistemas Integrados e Concorrentes (LAICO) em uma máquina com uma *CPU Intel i7-9700 3.00Hz* e *GPU RTX 2060* com 6 *GB* de memória, 1920 núcleos, *clock* de 1680 *MHz* e arquitetura *Turing*.

Para os testes, foram utilizados dois pares de sequências: o conjunto formado por *Corynebacterium efficien YS-314 DNA* e *Corynebacterium glutamicum ATCC 13032*, com sequências de aproximadamente 3 *MB*, e o conjunto formado pelos *Bacillus anthracis str. Ames* e *Rhodopirellula baltica SH 1 chromosome*, com sequências de aproximadamente 7 *MB*.

Em cada execução, avaliamos: (i) o tempo total gasto com todas as chamadas do sistema de *checkpoints* (*overhead*), (ii) o tempo total de execução, (iii) o impacto relativo do *overhead* no tempo total e (iv) o número de *checkpoints* realizados, conforme apresentado na Tabela 5.3.

Tabela 5.1: Sequências Comparadas

Tamanho	Nome	*Accession*
3.147 *MB*	*Corynebacterium efficien YS-314 DNA*	BA000035.2
3.282 *MB*	*Corynebacterium glutamicum ATCC 13032*	BX927147.1
5.522 *MB*	*Bacillus anthracis str. Ames*	NC_003997.3
7.145 *MB*	*Rhodopirellula baltica SH 1 chromosome*	NC_005027.1

5.2 Resultados da Execução

A execução do *CUDAlign* com *checkpoints* locais depende do parâmetro `-disk-size`, que indica quantos *megabytes* de informação poderão ser armazenados durante uma execução. A partir desse valor é determinado quantas vezes o `SpecialRowPartition` será ativado e, consequentemente, quantos *checkpoints* o programa irá gerar, ainda que sejam armazenados, no máximo, dois *checkpoints* simultaneamente.

Para cada um desses pares, o programa foi executado com `-disk-size=5000M` e `-disk-size=10000M`, resultando em quatro grupos de comparação.

Tabela 5.2: Resultados comparação das sequências de 3 *MB*

Comparação BA000035.2 com BX927147.1				
disk-size	*Overhead*	Tempo Execução	Impacto	Nº *Checkpoints*
5.000MB	3.9092s	1m11s	5,5059%	198
10.000MB	9.4519s	1m21s	11,6690%	384

Tabela 5.3: Resultados comparação das sequências de 7 *MB*

Comparação NC_003997.3 com NC_005027.1				
disk-size	*Overhead*	Tempo Execução	Impacto	Nº *Checkpoints*
5.000MB	3.6256s	4m09s	1,4560%	124
10.000MB	6.2564s	4m08s	2,5227%	182

A partir dos dados gerados nas execuções, observa-se que o número de *checkpoints* aumenta de acordo com o valor configurado em `-disk-size`. Isso ocorre porque, quanto maior o `disk-size`, menor o intervalo, em número de linhas, entre *checkpoints*, o que resulta na geração de mais *checkpoints* ao longo da execução.

Como cada arquivo de linha é salvo célula a célula a cada chamada de escrita do `SpecialRowPartition`, o tamanho dos *checkpoints* é impactado pelo tamanho das linhas, isto é, pelo tamanho das sequências avaliadas, o que implica em um número maior de operações de escrita para cada *checkpoint* salvo.

O aumento do número de *checkpoints* salvos teve impacto direto no *overhead* observado, inclusive para sequências menores. Isso se deve ao fato de que, ao término do salvamento de um *checkpoint*, é executado um conjunto adicional de operações, como a identificação dos arquivos que devem ser removidos e a sua efetiva exclusão do sistema de arquivos.

Por fim, quanto maior o comprimento das sequências, maior o tamanho de cada arquivo de checkpoint. Por conta disso, para um mesmo valor passado no `disk-size`, menos checkpoints são salvos, o que resulta em uma diminuição no overhead.

5.3 Teste do *Checkpoint*

Após salvar um arquivo de *checkpoint*, foi realizado um teste de integridade para verificar se o conteúdo armazenado estava correto. Como esse teste impõe por si só um *overhead* elevado, ele foi executado em um momento distinto do restante dos experimentos. Ao término da escrita de uma linha, é feita a desserialização do conteúdo do arquivo binário de volta para a *struct* `cell_t`. Como as linhas têm tamanho fixo e conhecido para cada execução do *CUDAlign*, é possível verificar se o que foi lido, e consequentemente o que havia sido escrito, está no formato correto.



Figura 5.1: Verificação de formato do *checkpoint* salvo

Capítulo 6

Conclusão

Este trabalho apresentou o projeto e a implementação de um mecanismo de *checkpoint* local para o *FT-CUDAlign*, voltado a reduzir o retrabalho causado por falhas durante a execução de algoritmos exatos de alinhamento de sequências biológicas em *GPU*. Diferentemente do mecanismo tradicional baseado apenas em *checkpoints* globais, que registra o estado da computação apenas nos *breakpoints* de cada iteração, o método proposto permite registrar periodicamente a última linha finalizada por cada instância do *CUDAlign*. Com isso, o sistema passa a dispor de pontos de restauração mais granulares, capazes de preservar quase todo o progresso realizado desde o último *breakpoint*.

A solução foi implementada de forma modular por meio das classes `SpecialRowsPartition` e `SpecialRowWriter`, sem alterar o fluxo principal do *FT-CUDAlign*. O mecanismo utiliza arquivos temporários e operações de renomeação atômica para garantir integridade mesmo em caso de falhas durante a escrita, mantendo no máximo dois *checkpoints* válidos por *GPU*. Os resultados experimentais demonstram que a sobrecarga adicional é compatível com a operação contínua do sistema: para sequências de aproximadamente $3MB$, o *overhead* variou de 3,9 s a 9,5 s, representando de 5,5% a 11,7% do tempo total de execução, dependendo da frequência de salvamento configurada. Para sequências maiores, em torno de 7 *MB*, o *overhead* observado foi de 3,6 s a 6,3 s, com um impacto relativo menor, entre 1,5% e 2,5%. Esses valores confirmam que o mecanismo de salvamento não compromete significativamente o desempenho da ferramenta e se integra de maneira eficiente ao fluxo de cálculo principal.

Além do mecanismo de *checkpoint* local, este trabalho integrou ao *FT-CUDAlign* uma atualização estrutural baseada no artigo [16], que substituiu o uso de memória de textura por memória global para armazenar as sequências de entrada na *GPU*. Esta modificação foi necessária devido à descontinuação da *API* de textura no *CUDA* 10.1 e eliminou o limite de 134 *MB* que restringia o processamento de sequências longas. A atualização garante compatibilidade com versões recentes do *CUDA* e, conforme demonstrado no

artigo referenciado, melhora o desempenho para sequências acima desse limite, eliminando o *overhead* de particionamento.

Um ponto importante a ser destacado é que o *checkpoint* local é atualmente executado apenas quando o *CUDAlign* roda com uma única *GPU* e com a *flag* de configuração de tamanho de disco. No modo *Multi-GPU*, o qual a execução ocorre via *Balancer*/*Controller* com duas ou mais *GPUs*, o fluxo de execução não percorre o módulo *SpecialRowsPartition*, e portanto o mecanismo de *checkpoint* local ainda não é ativado. Essa limitação reflete o estágio atual da integração e evidencia a necessidade de estender o suporte para execuções distribuídas.

Como trabalhos futuros, destacam-se três eixos principais. O primeiro é a generalização do mecanismo para múltiplas *GPUs*, garantindo que cada instância do *CUDAlign*, mesmo dentro do ambiente coordenado pelo *Controller*, execute seu próprio ciclo de salvamento e limpeza de *checkpoints*. O segundo é a implementação de um mecanismo completo de recuperação a partir de um *checkpoint* local, permitindo que uma *GPU* retome o processamento exatamente a partir da última linha salva. Por último, criar um parâmetro referente ao número de *checkpoints* salvos dedicado para a funcionalidade dos *checkpoints* locais, de modo a permitir um controle mais preciso do número de *checkpoints* desejados e desvincular seu funcionamento do *–disk-size*, que é utilizado para definir o limite de armazenamento de diversas outras estruturas da ferramenta. Com essas extensões, o *FT-CUDAlign* poderá oferecer tolerância a falhas mais eficiente, granular e adequada a cenários de execução prolongada em larga escala.

Referências

[1] Myers, Eugene W. e Webb Miller: *Optimal alignments in linear space*. Bioinformatics, 4(1):11–17, 1988. viii, 1, 11, 12

[2] Oliveira Sandes, Edans Flavius de, Guillermo Miranda, Xavier Martorell, Eduard Ayguadé, George Teodoro e Alba Cristina M. A. de Melo: *Cudalign 4.0: Incremental speculative traceback for exact chromosome-wide alignment in gpu clusters*. IEEE Transactions on Parallel and Distributed Systems, 27(10):2838–2850, outubro 2016. viii, 14, 16, 19

[3] Oliveira Sandes, Edans Flávius de: *Algoritmos paralelos exatos e otimizações para alinhamento de sequências biológicas longas em plataformas de alto desempenho*. Tese de Doutoramento, Universidade de Brasília, Brasília, DF, Brasil, 2015. Tese de Doutorado. Disponível em: `https://repositorio.unb.br/handle/10482/20248`, acesso em: 2025-12-04. viii, 18

[4] Soares, Filipe Maia: *Comparação paralela de sequências biológicas em múltiplas gpus com mecanismo de tolerância a falhas*. Tese de Mestrado, Universidade de Brasília, março 2025. viii, 1, 20, 21

[5] Mount, David W.: *Bioinformatics: Sequence and Genome Analysis*. Cold Spring Harbor Laboratory Press, second edição, 2004. 1, 5

[6] Smith, T.F. e M.S. Waterman: *Identification of common molecular subsequences*. Journal of Molecular Biology, 147(1):195–197, 1981, ISSN 0022-2836. `https://www.sciencedirect.com/science/article/pii/0022283681900875`. 1, 6

[7] Gotoh, Osamu: *An improved algorithm for matching biological sequences*. Journal of Molecular Biology, 162(3):705–708, 1982, ISSN 0022-2836. `https://www.sciencedirect.com/science/article/pii/0022283682903989`. 1, 8

[8] Oliveira Sandes, Edans Flavius de e Alba Cristina Magalhães Alves 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, maio 2013. 1, 17

[9] Almeida, André Atanasio Maranhão: *Novas abordagens para o problema do alinhamento múltiplo de sequências*. Tese de Doutoramento, Universidade Estadual de Campinas, Campinas, SP, Brasil, 2013. Tese de Doutorado. Disponível em: `https://hdl.handle.net/20.500.12733/1620274`. 3

[10] Durbin, Richard, Sean R. Eddy, Anders Krogh e Graeme Mitchison: *Biological Sequence Analysis: Probabilistic Models of Proteins and Nucleic Acids*. Cambridge University Press, Cambridge, UK, 1998. 4

[11] CP-Algorithms: *Introduction to dynamic programming*. `https://cp-algorithms.com/dynamic_programming/intro-to-dp.html`. 5

[12] Hirschberg, Daniel S.: *A linear space algorithm for computing maximal common subsequences*. Communications of the ACM, 18(6):341–343, 1975. 11

[13] O. Sandes, Edans Flavius de e Alba Cristina Magalhães Alves de Melo: *Smith-waterman alignment of huge sequences with gpu in linear space*. Em *2011 IEEE International Parallel & Distributed Processing Symposium (IPDPS)*, páginas 1199–1211, Anchorage, AK, USA, 2011. IEEE. 14

[14] O. Sandes, Edans Flavius de, Guillermo Miranda, Alba Cristina M. A. de Melo, Xavier Martorell e Eduard Ayguadé: *Cudalign 3.0: Parallel biological sequence comparison in large gpu clusters*. Em *2014 14th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid)*, páginas 160–169, Chicago, IL, USA, 2014. 18

[15] Figueiredo, Marco, João P. Navarro, Edans F. O. Sandes, George Teodoro e Alba C. M. A. Melo: *Parallel fine-grained comparison of long dna sequences in homogeneous and heterogeneous gpu platforms with pruning*. IEEE Transactions on Parallel and Distributed Systems, 32(12):3053–3065, dezembro 2021. 19

[16] Aragão, Rafael Rodrigues Gamas Domingos de: *Atualização do uso de memória de gpu na ferramenta masa-cudalign de comparação de sequências biológicas longas*. 2025. Disponível em: `https://bdm.unb.br/handle/10483/42129`. 26, 29, 33

Anexo I

Artigo apresentado no WPOS 2025

Checkpoint local e não coordenado para a aplicação paralela MASA-CUDAlign de comparação de sequências biológicas

*

Gustavo Lopes Dezan
Departamento de Ciência da Computação
Universidade de Brasília
Brasília, Brasil
202033463@aluno.unb.br

Isabel Pierre Starling
Departamento de Ciência da Computação
Universidade de Brasília
Brasília, Brasil
202006420@aluno.unb.br

Alba Cristina Magalhaes Alves de Melo
Departamento de Ciência da Computação
Universidade de Brasília
Brasília, Brasil
alves@unb.br

***Abstract*—Biological sequence comparison is a fundamental task in bioinformatics, enabling the identification of mutations, functional similarities, and evolutionary relationships among DNA, RNA, and protein sequences. Exact alignment algorithms such as Smith–Waterman and Gotoh provide highly accurate results but require intensive computational resources, with time and space complexity proportional to the product of sequence lengths. To address these limitations, MASA-CUDAlign leverages CUDA-based massive parallelism across multiple GPUs, significantly reducing execution time while preserving accuracy. Building upon this framework, FT-CUDAlign introduces fault tolerance, allowing computation to recover from hardware or communication failures using a checkpointing mechanism. However, its current checkpoint strategy, based on predefined breakpoints, can result in substantial loss of progress when failures occur between checkpoints.**

This work proposes an enhanced checkpointing mechanism for FT-CUDAlign that implements local checkpoints at the GPU level. The new method periodically saves the most recently processed row using atomic write operations and controlled deletion of outdated checkpoint files, ensuring data integrity even under unexpected interruptions. This improvement minimizes recomputation overhead, increases fault recovery efficiency, and enhances the overall robustness of large-scale parallel sequence alignment on multi-GPU systems.

***Index Terms*—Bioinformatics, Sequence Alignment, Smith–Waterman Algorithm, GPU Computing, CUDA, Fault Tolerance, Checkpoint**

I. Introdução

A comparação de sequências biológicas é uma das tarefas centrais da bioinformática, sendo essencial para compreender relações evolutivas, identificar mutações e inferir funções moleculares. Essa análise consiste em avaliar o grau de similaridade entre cadeias de DNA, RNA ou proteínas, permitindo identificar regiões homólogas que podem refletir origem ou função comuns.

Nesse contexto, os algoritmos de alinhamento, como os de Smith-Waterman [2] e Gotoh [3], são amplamente utilizados por sua precisão na obtenção de resultados ótimos. Entretanto, o custo computacional desses métodos é elevado, apresentando complexidade proporcional ao produto dos comprimentos das sequências. Com isso, surgiu o MASA-CUDAlign que explora a arquitetura CUDA para realizar o alinhamento de forma massivamente paralela em múltiplas GPUs, reduzindo o tempo de execução sem comprometer a exatidão dos resultados.

Com o intuito de aumentar a robustez desse sistema, foi desenvolvido o FT-CUDAlign, uma extensão do MASA-CUDAlign que incorpora tolerância a falhas. Essa funcionalidade permite que o sistema se recupere de falhas de hardware ou comunicação sem a necessidade de reiniciar completamente a execução, utilizando mecanismos de checkpoint responsáveis por armazenar periodicamente o estado do processamento. A abordagem atualmente empregada baseia-se em breakpoints, os quais consistem em pontos específicos do alinhamento onde o estado é salvo, o que implica que falhas ocorridas após longos intervalos desde o último checkpoint resultam em perda significativa de progresso e aumento do tempo de reprocessamento.

Visando mitigar essa limitação, este trabalho propõe uma melhoria no mecanismo de checkpoint do FT-CUDAlign, introduzindo um método de salvamento da última linha processada. Essa abordagem tem como objetivo reduzir a quantidade de trabalho perdida em caso de falhas, aumentando a eficiência do processo de recuperação. O novo mecanismo adota checkpoints locais por GPU, implementando operações de escrita atômicas e remoção controlada de arquivos (checkpoints)

antigos, de modo a garantir a consistência e a integridade dos dados armazenados mesmo diante de interrupções inesperadas.

II. Comparação de Sequências

Na comparação de sequências são utilizados algoritmos que aplicam sistemas de pontuação capazes de quantificar o grau de similaridade entre as sequências. Esses sistemas consideram correspondências exatas (*matches*), substituições (*mismatches*) e lacunas (*gaps*), que representam inserções ou deleções. A forma de penalizar os *gaps* pode seguir diferentes modelos: no modelo linear, cada abertura ou extensão tem custo fixo, já no modelo de *affine gap*, distingue-se o custo de abertura (*gap_open*) do custo de extensão (*gap_extend*), o que reflete de forma mais realista o comportamento biológico.

Existem dois principais tipos de alinhamento: o *global* e o *local* (Fig 1. O alinhamento global busca o melhor alinhamento entre duas sequências em sua totalidade. Já o alinhamento local identifica apenas as regiões de maior similaridade, ignorando partes não relacionadas. Ambos os métodos baseiam-se em *programação dinâmica*, uma técnica que divide o problema em subproblemas menores e utiliza matrizes de pontuação para garantir soluções ótimas.

Sequência 1: A T T C T G C G A T A C G
Sequência 2: C T G G A T T C A A

Alinhamento global:

```
A T T C T G C G A T A C G
- - - | | | - - - - | - - -
- - - C T G G A T T C A A
```

Alinhamento local:

```
- T T C T G C G A T - - -
- - - - - - - - | | | - - -
- - - - - - - G G A T T - -
```

Fig. 1. Alinhamento global e local entre duas sequências biológicas.

Entre os algoritmos exatos mais relevantes destaca-se o *Smith-Waterman*, amplamente utilizado para o alinhamento local de sequências biológicas. Esse algoritmo utiliza a técnica de *programação dinâmica* para construir uma matriz bidimensional H, onde cada elemento $H[i][j]$ representa o melhor *score* de alinhamento local entre os prefixos $A[1..i]$ e $B[1..j]$. A matriz é preenchida de acordo com a equação de recorrência:

$$H_{i,j} = \max \begin{cases} 0 \\ H_{i-1,j-1} + s(A[i], B[j]) \\ H_{i,j-1} + \text{gap} \\ H_{i-1,j} + \text{gap} \end{cases} \quad (1)$$

Aonde $s(A[i], B[j])$ retorna uma pontuação positiva em caso de *match* e negativa em caso de *mismatch*, enquanto o parâmetro *gap* representa a penalidade associada à abertura ou extensão de uma lacuna. A presença do valor zero assegura que apenas regiões de similaridade contribuam para o alinhamento, característica essencial do alinhamento local. Após o preenchimento da matriz, o melhor alinhamento é obtido por meio do processo de *traceback*, que segue o caminho inverso a partir do maior valor encontrado. O algoritmo apresenta complexidade $O(m.n)$ tanto em tempo quanto em espaço, onde m e n são os comprimentos das sequências, tornando-o computacionalmente intensivo para genomas de grande porte.

Algoritmos derivados aprimoram o desempenho e a flexibilidade do método. O *Gotoh* estende o *Smith-Waterman* ao incorporar penalidades do tipo *affine gap*, diferenciando o custo de abertura e de extensão de lacunas. Já o *Myers-Miller* [1] propõe uma otimização em espaço linear, aplicando a técnica de *divide and conquer* para reduzir o consumo de memória sem comprometer a obtenção da solução ótima.

Diante das limitações computacionais dos métodos exatos de alinhamento, o MASA-CUDAlign se destaca como uma solução de referência em alto desempenho para esse tipo de processamento. A ferramenta utiliza CUDA para realizar processamento paralelo em múltiplas GPUs, distribuindo a carga de trabalho e reduzindo significativamente o tempo de execução sem comprometer a precisão do alinhamento.

III. FT-CUDAlign

Tolerância a falhas refere-se à capacidade de um sistema de retomar sua execução após falhas, ao custo de lidar com um overhead inerente às técnicas utilizadas

Dentre as modalidades de *checkpoint*, destacam-se os **coordenados** e os **não coordenados**. Nos coordenados, o sistema sincroniza os processos para que todos gravem o estado em um intervalo comum, produzindo um checkpoint global consistente, ao custo de um maior overhead. Nos não coordenados, cada processo realiza seu *checkpoint* de forma independente, o que favorece a escalabilidade, mas aumenta o risco de inconsistências e do efeito cascata durante a recuperação.

O FT-CUDAlign [5] é uma extensão do MASA-CUDAlign, projetada para incorporar tolerância a falhas (*Fault Tolerance*) em sistemas de alinhamento massivamente paralelos executados em múltiplas GPUs. Seu principal objetivo é garantir a continuidade da execução mesmo diante de falhas de hardware, erros de comunicação ou desconexões de GPUs, minimizando a necessidade de reiniciar toda a computação.

O sistema é dividido em três fases principais:

1) `Detecção Pré-Execução`, em que o Controller verifica a disponibilidade das GPUs antes de iniciar a execução;
2) `Detecção Durante a Execução`, que monitora continuamente a comunicação entre as GPUs para identificar falhas;
3) `Procedimento de Recuperação`, responsável por restabelecer conexões, reconfigurar a distribuição de tarefas e retomar a execução a partir de um checkpoint.

Durante a execução, as GPUs operam de forma encadeada, comunicando-se apenas com suas vizinhas imediatas anterior e posterior para trocar as colunas de borda responsáveis por manter a continuidade da matriz de alinhamento. A execução é dividida em intervalos denominados *breakpoints*, que marcam pontos específicos do progresso do alinhamento e servem

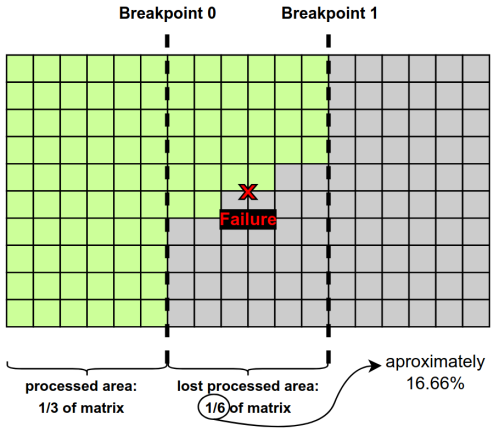


Fig. 2. Exemplificação dos checkpoints globais

como referência para a criação de *checkpoints*. O processo de *checkpoint* é coordenado por um módulo denominado *Controller*, que, a cada *breakpoint*, solicita às GPUs o envio de seus dados de fronteira (*border data*) e os armazena de forma consistente. Esses dados formam o estado global do sistema, permitindo que a execução seja retomada de maneira segura a partir desse ponto em caso de falha, conforme a (Fig. 3).

Essa abordagem garante a integridade do processamento, porém apresenta uma limitação importante: caso uma falha ocorra após um longo período desde o último *breakpoint*, o sistema perde todo o progresso realizado desde o último *checkpoint*, pois o *FT-CUDAlign* salva o estado apenas nesses pontos de interrupção. Assim, quanto maior o intervalo entre *breakpoints*, maior será a quantidade de trabalho descartada e o custo de reprocessamento durante a recuperação. Apesar desse custo potencial, o mecanismo de *checkpointing* baseado em *breakpoints* oferece uma solução prática e eficiente para ambientes de execução prolongada e sujeitos a falhas.

IV. Projeto do Mecanismo

O objetivo de introduzir *checkpoints* locais de linha no projeto do FT-CUDAlign é reduzir a taxa de perdas e, consequentemente, o reprocessamento no mecanismo de recuperação de falhas.

Para isso, o CUDAlign salva, além das colunas de borda, linhas de interesse a cada certo número de linhas processadas. Como cada processo do CUDAlign opera sobre uma fatia da matriz, esses *checkpoints* podem ser não coordenados entre os processos em execução. Isso garante que cada bloco (processado por uma GPU) tenha seu próprio *checkpoint* de linha, permitindo que, em caso de falha, um nova execução seja iniciada a partir da última linha salva.

Para salvar um *checkpoint* local, cada instância do CUDAlign executa os passos a seguir (Fig. 3):

1) `SpecialRowsPartition` chama o método `write` de `SpecialRowWriter` para persistir a última linha em um novo arquivo;

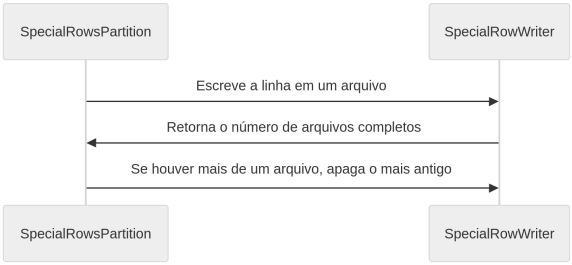


Fig. 3. Diagrama de sequência do checkpoint local de linha

2) `SpecialRowWriter` retorna a quantidade de arquivos na pasta de *checkpoints* do CUDAlign;
3) se existirem pelo menos dois arquivos, remove o mais antigo, mantendo apenas o mais recente.

Um checkpoint de linha é um arquivo binário com o conteúdo de uma linha completa. Cada processo do CUDAlign tem seu próprio diretório de *checkpoints* e dentro dele cada arquivo é refente a uma linha. A seleção do arquivo mais antigo é feita de acordo com o tempo da última escrita de cada um. Assim, como cada instância escreve apenas um arquivo por vez e depois de concluído ele não é mais atualizado, é garantido que o arquivo com tempo de escrita mais recente é sempre referente a uma linha de índice maior.

O processo de escrita do arquivo é realizado pelo `SpecialRowWriter`, que escreve uma célula da matriz de cada vez em um arquivo temporário e só quando a linha está completa faz uma operação de rename atômica, de modo que somente após essa operação o arquivo passa a contar como um checkpoint completo. Isso garante que no caso de ocorrer uma falha durante a escrita de uma linha, o checkpoint anterior a ela ainda será o mais recente.

Além disso, a operação de apagar os arquivos antigos ocorre ao final do processo de escrita de uma nova linha e verifica quantos arquivos não temporários (*checkpoints*) existem, apagando todos até deixar o mais recente. Na prática, esse procedimento é chamado a cada novo checkpoint e apaga o arquivo mais antigo, de modo que o sistema alterne entre 1 e 2 arquivos de checkpoint salvos, mas nunca menos do que isso (exceto antes da escrita do primeiro *checkpoint*).

Como cada GPU processa uma área independente da matriz, compartilhando apenas o conteúdo da coluna de borda, assume-se que no caso de uma GPU falhar, as máquinas anteriores a ela continuarão a execução até que seu buffer fique cheio, caso em que elas começam a esperar pelos dados da GPU em recuperação, e as posteriores param esperando pelas mensagens da máquina em falta. Esta por sua vez, após recuperar o estado do último checkpoint de linha, retoma o processamento e a enviar e consumir informações, desencadeando assim o retorno do processamento do FT-CUDAlign. Por meio dessa abordagem, os *checkpoints* locais

nessa implementação não correm risco de sofrer o efeito cascata, uma vez que eles serão utilizados de modo isolado para a recuperação do bloco selecionado.

V. Resultados Experimentais

Os testes foram realizados no Laboratório de Sistemas Integrados e Concorrentes (LAICO) em uma máquina com GPU RTX 2060 com 6GB de memória, 1920 núcleos, clock de 1680 MHz e arquitetura Turing.

Nesse cenário, foi avaliado o overhead introduzido pelo mecanismo de checkpoints locais medindo seu tempo total e comparando com o tempo total da execução do FT-CUDAlign apenas com os checkpoints globais. O tempo overhead encontrado foi de aproximadamente 2 segundos por execução.

TABLE I
Comparação de desempenho

Tamanho da sequência	Tempo s/ ckpt (aprox.)	Overhead abs.	Overhead rel.
3 MB	≈ 60 s	≈ 2 s	3,33%
7 MB	≈ 240 s	≈ 2 s	0,8%

VI. Conclusão

O FT-CUDAlign é uma extensão do MASA-CUDAlign que introduz tolerância a falhas em execuções distribuídas em múltiplas GPUs, permitindo a retomada do processamento após falhas de hardware ou comunicação. No entanto, seu mecanismo atual de *checkpoint*, baseado em *breakpoints* globais, pode ocasionar perda significativa de progresso quando uma falha ocorre entre dois pontos de salvamento.

Este trabalho apresentou o projeto e a implementação de um novo mecanismo de *checkpoint* local no FT-CUDAlign, que realiza salvamentos por GPU. O método introduz o conceito de *checkpoints* de linha, nos quais a última linha processada é periodicamente salva em disco por meio de operações de escrita e remoção controlada de arquivos antigos. Essa abordagem reduz a dependência de pontos fixos de salvamento e garante a integridade dos dados mesmo em caso de interrupções inesperadas.

Nesta etapa, o escopo do trabalho concentra-se exclusivamente na criação e gerenciamento dos checkpoints locais, sem ainda abranger o processo de recuperação. Como trabalhos futuros, pretende-se implementar o mecanismo de leitura e restauração desses checkpoints, permitindo que o FT-CUDAlign retome a execução exatamente a partir do ponto interrompido. Isso possibilitará avaliar, de forma quantitativa, a redução de tempo de reprocessamento e o aumento da eficiência geral do sistema frente a falhas.

References

[1] E. W. Myers and W. Miller, "Optimal alignments in linear space," *Bioinformatics*, vol. 4, no. 1, pp. 11–17, 1988.

[2] T. F. Smith and M. S. Waterman, "Identification of common molecular subsequences," *J. Mol. Biol.*, vol. 147, no. 1, pp. 195–197, 1981.

[3] O. Gotoh, "An improved algorithm for matching biological sequences," *J. Mol. Biol.*, vol. 162, no. 3, pp. 705–708, 1982.

[4] E. F. O. Sandes, G. Miranda, X. Martorell, E. Ayguadé, G. Teodoro, and A. C. M. Melo, "CUDAlign 4.0: Incremental speculative traceback for exact chromosome-wide alignment in GPU clusters," *IEEE Trans. Parallel Distrib. Syst.*, vol. 27, no. 10, pp. 2838–2850, 2016.

[5] F. M. Soares, "Comparação Paralela de Sequências Biológicas em Múltiplas GPUs com Mecanismo de Tolerância a Falhas," M.Sc. dissertation, Dept. of Computer Science, Universidade de Brasília (UnB), Brasília, Brazil, Mar. 12, 2025.