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

Utilização de Técnicas e Instruções Especiais para Acelerar o Casamento de Padrões Exato e Aproximado em GPU

Lucas S. N. Nunes

Dissertação apresentada como requisito parcial para
conclusão do Mestrado em Informática

Orientador
Prof. Dr. Jacir Luiz Bordim

Brasília
2018

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

Utilização de Técnicas e Instruções Especiais para Acelerar o Casamento de Padrões Exato e Aproximado em GPU

Lucas S. N. Nunes

Dissertação apresentada como requisito parcial para
conclusão do Mestrado em Informática

Prof. Dr. Jacir Luiz Bordim (Orientador)
CIC/UnB

Prof. Dr. Ricardo Pezzuol Jacobi
Universidade de Brasília

Prof. Dr. Daniel Sundfeld Lima
Instituto Federal de Brasília

Prof. Dr. Bruno Luiggi Macchiavello Espinoza
Coordenador do Programa de Pós-graduação em Informática

Brasília, 31 de julho de 2018

Dedicatória

Dedico este trabalho a meus pais e a todos que direta ou indiretamente contribuíram de alguma maneira para sua realização.

Agradecimentos

Agradeço ao meu orientador, professor Jacir Luiz Bordim por toda a dedicação, paciência, amizade e apoio que foram cruciais para a realização deste trabalho. Agradeço também a banca examinadora Prof. Jacobi e o Prof. Sundfeld por terem dedicado seu tempo em avaliar este trabalho, sempre com o objetivo de aperfeiçoá-lo.

Resumo

Placas gráficas evoluíram significativamente no decorrer dos últimos anos, principalmente no que tange a capacidade de processamento, e se tornaram uma ferramenta essencial para realizar tarefas que permitam um certo grau de paralelismo. A *Graphics Processing Unit* (GPU) é um circuito projetado para processamento gráfico, ela possui hoje núcleos de processamento na ordem de centenas. Nos últimos anos a GPU vêm sendo cada vez mais utilizada para processamento de propósito geral, o que chamamos de *General-purpose Computing on Graphics Processing Units* (GPGPU). Estas placas vêm sendo utilizadas na comunidade científica em várias áreas, tais como, criptografia, ordenação, grafos e alinhamento de sequências. A proximidade de dois padrões é uma medida importante para várias aplicações, incluindo bioinformática e processamento de sinais. Este trabalho busca acelerar a busca por casamento de padrões através de uma funcionalidade em placas recentes que permite o uso eficiente da comunicação entre um conjunto de *threads* que rodam concorrentemente em um mesmo *Streaming Multiprocessor* (SM). Em uma primeira proposta utilizamos esta comunicação e obtivemos ganhos maiores que 2,5 vezes em relação a uma alternativa proeminente, em uma segunda proposta otimizamos o uso das comunicações e conseguimos ganhos maiores que 1,3 em relação a primeira proposta. Por fim propomos uma alternativa ao Rabin-Karp para o casamento de padrões exato. Essa alternativa consiste em utilizar soma de prefixos para poder paralelizar de forma otimizada o algoritmo, além disso conseguimos comparar vários padrões de uma vez sem uma diferença significante no tempo. Alcançamos ganhos maiores que 2 vezes para um padrão e maiores que 10 vezes para 256 padrões.

Palavras-chave: Casamento de padrões exato, Casamento de padrões aproximado, distância de edição, GPGPU, CUDA, Rabin-Karp.

Abstract

Graphics card have evolved significantly over the last few years, especially in terms of processing capacity, and have become an essential tool for performing tasks that allow a degree of parallelism. *Graphics Processing Unit* (GPU) is a circuit designed for graphics processing, it has processing cores in the order of hundreds. In recent years, GPU has been increasingly used for general-purpose processing, which we call *General-purpose Computing on Graphics Processing Units* (GPGPU). These boards have been used in the scientific community in several areas, such as cryptography, ordering, graphs and sequence alignment. The closeness of a match is an important measure with a number of practical applications, including computational biology and signal processing. This work is focused on accelerate the string matching through a feature in recent boards, the efficient use of communication between a group of threads which run concurrently in the same *Streaming Multiprocessor* (SM), Using this communication we have achieved in a first proposal gains greater than 2.5 in relation to a prominent alternative, in a second proposal we optimize the use of communication and we achieved gains greater than 1.3 in relation to the first proposal. Finally, we propose an alternative to Rabin-Karp for the string matching, this alternative consists in using prefix-sum to maximize the parallelziation of the algorithm, in addition we can compare several patterns at once without a significant difference in time,we obtain as a result gains greater than 2 for one pattern and greater than 10 for 256 patterns.

Keywords: String matching, Approximate string matching, Edit distance, GPGPU, CUDA, Rabin-Karp.

Sumário

Lista de Figuras

Lista de Tabelas

Lista de Abreviaturas e Siglas

ASM *Approximate String Matching.*

CC *Compute Capability.*

CPU *Central Processing Unit.*

DE Distância de Edição.

DP *Double Precision.*

GPGPU *General-purpose Computing on Graphics Processing Units.*

GPU *Graphics Processing Unit.*

MP *Multiprocessor.*

SIMD *Single Instruction Multiple Data.*

SM *Streaming Multiprocessor.*

SP *Stream Processor.*

Capítulo 1

Introdução

Placas gráficas evoluíram consideravelmente no decorrer dos últimos anos, principalmente no que tange a capacidade de processamento, e se tornaram uma ferramenta essencial para realizar tarefas que permitem um certo grau de paralelismo [1]. O uso do potencial dessas placas para processamento geral vem permitindo trazer eficiência na utilização de vários algoritmos na literatura [2] [3] [4].

Como os fabricantes estão permitindo cada vez mais linguagens de alto nível para utilização dessas placas, o uso delas para propósito geral vem ganhando espaço. Encontrar sequências de caracteres iguais a um certo padrão tem um número grande de aplicações possíveis, como ordenação [5] [6], compressão de dados [7], criptografia [8], grafos [9], *deep learning* [10], casamento de padrões [4] e comparação de sequencias de DNA [11] [12] [13] [14]. Além das funções de processamento gráfico, as GPUs também são capazes de executar algoritmos de propósito geral, chamado *General-purpose Computing on Graphics Processing Units* (GPGPU).

Transferência de dados vem sendo um gargalo na performance da computação [15], até algum tempo GPUs convencionais só possuíam dois métodos de comunicação entre *threads*, a memória compartilhada e a memória global. Contudo a utilização de comandos *shuffle* que permitem que uma *thread* acesse um registrador de outra *thread* desde que elas estejam sendo executadas concorrentemente em um mesmo *Streaming Multiprocessor* (SM), esse grupo de *threads* chamada de *warp*, permite uma transferência de dados mais eficiente [15] [16].

Com a utilização das instruções *shuffle* aliada a outras técnicas conseguimos resultados melhores em trabalhos referentes a casamento de padrões [17] [18] [19] [20], nossa proposta visa explorar alternativas para acelerar este casamento.

1.1 Objetivos

Este trabalho tem como objetivo principal avaliar o emprego de técnicas de comunicação inter-warp com alternativa para acelerar o casamento de padrões exato e aproximado. Para atingir o objetivo principal, os seguintes objetivos específicos são definidos:

1. Utilizar técnicas validadas em outros trabalhos a fim de melhorar o desempenho do algoritmo proposto.
2. Definir métricas e cargas para coletar resultados.
3. Analisar o desempenho do algoritmos.

1.2 Contribuições

Como contribuição do mestrado obtivemos duas publicações no CANDAR (*International Symposium on Computing and Networking*), classificado como qualis B2 e uma publicação no periódico *IEICE transactions on information and systems* classificado como qualis B1. Seguem os trabalhos publicados:

- *A Memory-Access-Efficient Implementation of the Approximate String Matching Algorithm on GPU* [18].
- *A Memory-access-efficient Implementation for Computing the Approximate String Matching Algorithm on GPU* [20].
- *A Fast Approximate String Matching Algorithm on GPU* [19].

1.3 Estrutura do documento

No Capítulo 2 vemos os principais conceitos sobre GPU, suas arquiteturas e classificações. Apresenta a arquitetura de programação de placas Nvidia, são mostrados os principais componentes e funcionalidades da arquitetura CUDA. O Capítulo 3 apresenta o estado da arte sobre algoritmos de reconhecimento de padrão em texto. O problema de casamento de cadeias de caracteres tanto exato quanto com busca aproximada é abordada nesse capítulo. Também é apresentado uma implementação ótima em tempo de execução de casamento de caracteres com busca aproximada, implementado em GPU. O Capítulo 4 apresenta nossa proposta para casamento de padrões aproximado, nele é descrita a técnica e o algoritmo utilizados, bem como os resultados. O capítulo 5 apresenta nossa proposta de Casamento de Padrões exato de forma paralela, esta utiliza como base o Rabin-Karp.

Capítulo 2

Unidade de Processamento Gráfico de Propósito Geral

Neste capítulo, serão introduzidos conceitos sobre unidade de processamento gráfico para propósitos gerais (do Inglês, *General-purpose Computing on Graphics Processing Units* (GPGPU)). A Seção 2.1 apresenta os principais componentes de GPUs Nvidia. A Seção 2.2 explica a plataforma CUDA encontrada em uma GPU Nvidia e uma breve introdução sobre evolução das placas gráficas.

2.1 Arquitetura e Plataforma de Computação Paralela

Uma GPU é um *hardware* projetado inicialmente para processamento gráfico e processamento geral. Nessa seção veremos seus componentes mais importantes e o *software* que envolve soluções CUDA.

2.1.1 Componentes de *Hardware*

A Figura 2.1 mostra um *Streaming Multiprocessor* (SM) de uma gpu Nvidia. A figura mostra que cada SM possui dois blocos de processamento com 32 *cores* cada, estes 32 *cores* dão origem ao que chamamos de *warp*. Os cálculos de ponto flutuante são evoluíram nesta arquitetura, existem 16 *Double Precision* (DP) núcleos em cada bloco de processamento. No modelo GP100 temos 60 SMs.

Figura 2.1: Exemplo de um *Streaming Multiprocessor* (SM) [21].

2.2 CUDA

CUDA é uma plataforma criada pela Nvidia que permite uma programação em GPGPU através de várias linguagens de programação como C, C++ e Fortran e de *frameworks* como OpenCL [22]. A arquitetura foi criada devido à necessidade de se utilizar placas de vídeo para propósitos gerais. Permite a utilização de componentes da placa como memória compartilhada ou constante em uma linguagem mais acessível,

Um programa em CUDA permite manipulação de dados tanto no hospedeiro (CPU) quanto no dispositivo(GPU). Quando a função é executada pela placa de vídeo é chamado de *kernel*. Na Figura 2.2 mostra a hierarquia de execução da arqtuiteura CUDA. As *threads* são organizadas em blocos os blocos são organizados em *grids*, os *grids* são compostos por *threads* que executam o mesmo kernel Em um *grid* podem existir vários blocos de execução, estes blocos são independentes e não pode ser assegurada nenhuma comunicação ou sincronia entre eles, dentro destes blocos podem ser executadas várias *threads*, estas por sua vez permitem comunicação e sincronia.

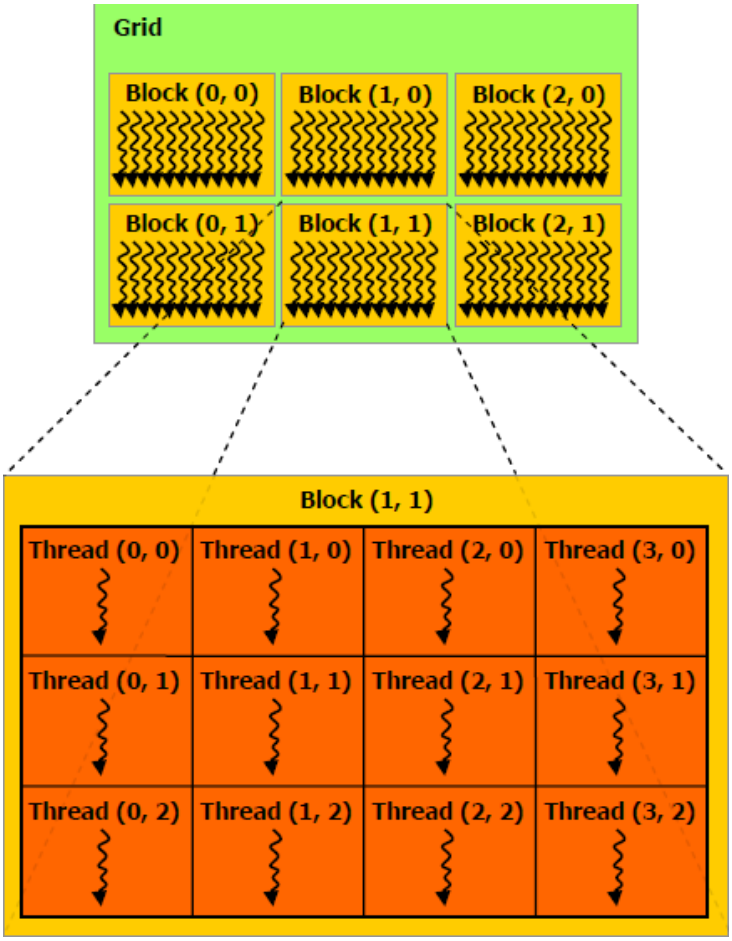


Figura 2.2: Organização de *threads*, blocos e *grids* [22].

2.2.1 *Threads*, Blocos e *Warps*

As *threads*, os blocos e os *grids* podem ser diferenciados pelo seu id. Este id pode ser obtido pelos comandos `threadIdx.d`, `blocIdx.d` e `blockDim.d` onde *d* representa a dimensão (x, y ou z) utilizada. Caso se esteja utilizando apenas uma dimensão e se deseja saber o id de uma *thread* por exemplo, o comando `treadIdx.x` retornará o id da *thread* que executa este comando.

Muitas vezes as *threads* precisam estar sincronizadas. Por exemplo, quando há a troca de dados entre *threads* é necessário saber se um determinado dado já está pronto. O comando `__syncthread()` força o sincronismo de *threads* de um mesmo bloco. Quando se usa esse comando a *thread* aguarda todas as outras pertencentes ao mesmo bloco chegarem nesse comando também. Dessa forma todas estarão sincronizadas e um acesso à memória compartilhada, por exemplo, poderá ser feito sem inconsistências. evitando condição de corrida. Observe que a ordem de execução ou mesmo sincronização entre blocos não pode ser determinada, logo estes blocos devem funcionar de forma independente.

Warps são um agrupamento de *threads*. Até o presente momento, o tamanho dessas

warps sempre foi de 32 *threads* em placas Nvidia [22]. Uma *warp* sempre tenta executar as instruções de suas *threads* simultaneamente. Quando duas ou mais *threads* tentam acessar um mesmo banco de memória esse acesso é feito em turnos.

2.2.2 Evolução do CUDA

Os primeiros computadores não se utilizavam de recursos gráficos, apenas era necessário mostrar caracteres na tela. Com o passar do tempo, os computadores começaram a utilizar interface gráfica. Logo após surgiram aplicações como jogos e edição de vídeo, o que causou a evolução das placas de vídeos existentes. A grande expansão de uso de GPUs para propósitos não gráficos vieram com a plataforma de programação CUDA [23]. Como mostrado no capítulo introdutório, a plataforma CUDA continua sendo bastante utilizada.

Atualmente, existem sete arquiteturas CUDA: a série G80(2006), a série GT200(2008), Fermi(2010), Kepler(2012), MaxWell(2014), Pascal(2016) e Volta(2017). Cada uma destas arquiteturas é classificada de acordo com o seu *Compute Capability* (CC). O CC é composto por dois números $x.y$, onde mudanças no número x representam uma grande diferença na arquitetura dessas placas e mudanças no número y representam modificações menores. Como classificação das arquiteturas citadas temos $1.y$ para as duas primeiras séries, $2.y$ para a Fermi, $3.y$ para a Kepler, $5.y$ para a Maxwell, 6.0 para a Pascal e 7.0 para a Volta.

A Tabela 2.1 mostra a comparação de algumas características importantes para programação de acordo com o CC. Por exemplo, mesmo que a versão 2.0 tenha 32K registradores de 32 bits para cada multiprocessador, cada *thread* só pode utilizar 63 destes registradores. Dessa forma a GPU deixaria de usar registradores e utilizaria a memória local se uma *thread* tivesse que utilizar mais de 63 registradores de 32 bits. A seguir será apresentado um breve histórico da evolução das GPUs Nvidia, estas placas podem ser separadas por séries e também pela sua CC.

Série G80

Essa série foi criada em novembro de 2006 nela veio a primeira GPU com suporte para linguagem C. Também foi a primeira série a ter um processador unificado e ter memória compartilhada. Essa série inovou com o processador que escalona *threads*, evitando que o programador tivesse que controlar isso manualmente.

Tabela 2.1: Evolução das placas Nvidia [22].

	Compute Capability					
Especificações técnicas	**1.0**	**2.0**	**3.0**	**5.0**	**6.0**	**7.0**
Tamanho da *warp*	32					
Número máximo de blocos por SM	8		16	32		
Número máximo de *warps* por SM	24	48	64			
Número máximo de *threads* por SM	768	1536	2048			
Número de registradores de 32 bits por SM	8K	32K	64K			
Número máximo de registradores de 32 bits por bloco	256					
Número máximo de registradores de 32 bits por *thread*	128	63	255			
Memória compartilhada total por SM	16KB	48KB		64KB		96KB
Número de bancos de memória compartilhada	16	32				
Tamanho da memória local por *thread*	16KB	512KB				

Série G200

Foi uma evolução da série G80, foi lançada em junho de 2008. A série G200 dobrou o número de núcleos (de 128 da série anterior, para 240). Cada processador teve seu tamanho de registradores duplicados, permitindo um número maior de *threads* a serem executadas ao mesmo tempo. Foi acrescentado a funcionalidade de utilização de ponto flutuante do tipo *double*, o que foi muito importante para processamento de alto desempenho. Nessa série foi introduzido o acesso simultâneo de um dado na memória por diversas *threads*, isto foi muito importante em eficiência.

Fermi

O grande passo dessa geração foi a elevação da performance em relação à série anterior. A utilização de ponto flutuante teve um grande salto em desempenho. Devido às restrições no acesso à memória compartilhada, criou-se uma hierarquia para acesso à memória de forma a resolver esta questão. Nesta arquitetura foram lançadas placas com 512 núcleos divididos em 16 SMs [24].

Kepler

Essa arquitetura trouxe um grande ganho em eficiência energética, conseguindo aumentar o processamento em mais de 3 vezes por watt em relação a Fermi. O número de registradores dobrou. Disponibilização de instruções que permite a troca de dados entre *threads* de uma mesma *warp* sem utilizar a memória compartilhada. Também foram adicionadas operações atômicas para acelerar o processamento [16].

Maxwell

Esta série tem uma performance energética 2 vezes melhor que a Kepler. O tamanho da cache L2 foi aumentada em 4 vezes e foram adicionadas funcionalidades do ponto de vista gráfico. Estas mudanças foram realizadas, pois a resolução de vídeo foi bastante aumentada nos anos antecessores [25].

Pascal

Arquitetura com foco na utilização de várias GPUs. Com a tendência de utilização de múltiplas GPUs por computador esta arquitetura tem uma interface de alta velocidade, NVLink, que provê transferências entre GPUs de até 160 Gigabytes por segundo de forma bidirecional [21]. A NVLink tem uma boa eficiência ao se ter que transferir muitos dados entre várias GPUs acopladas em um único computador. Algumas placas possuem HBM2 que triplicaram a velocidade de acesso a memória global.

Volta

Arquitetura mais recente da Nvidia, possui foco em Computadores de alto desempenho e data centers. A GPU V100 lançada nesta arquitetura possui $21,1$ bilhões de transistores e aumenta a performance de ponto flutuante e reduz a utilização de energia em 50% do SM comparado com a arquitetura Pascal. Além disso, ela possui a segunda geração do NVLink que permite transferência de até 300 Gigabytes por segundo [26].

2.2.3 Um Programa em CUDA

Um programa para GPUs deve ser bem elaborado e deve considerar a utilização dos diversos tipos de memória, pois cada uma tem suas características que influenciam significativamente o desempenho do programa.

CUDA pode ser utilizada com base em várias linguagens, entre elas temos C, C++, Fortran e Python [23]. Como exemplo mostramos a linguagem CUDA C, para se utilizar um *kernel* na GPU é necessário colocar a diretiva `__global__` na função, indicando que aquela função pode ser chamada pela *main* do programa e deverá ser executada na GPU. A chamada ao *kernel* deve conter pelo menos o número T de *threads* e B de blocos que serão utilizados.

No código apresentado no Algoritmo 2.1 é realizada a adição de vetores forma paralela. Podemos ver algumas diferenças em relação a um código em C puro. Para se utilizar a memória da GPU é necessário criar um ponteiro no lado da CPU e realizar um `cudaMalloc` para alocar o espaço na memória da GPU, como é mostrado nas linhas 18 a 20. Para copiar valores entre a CPU e a GPU é utilizado o comando `cudaMemcpy` que tem como

argumentos o ponteiro pra GPU, a variável da CPU, a quantidade de *bytes* a ser copiado e o sentido da cópia. Por exemplo, na linha 22 e 23 é utilizado o sentido da CPU pra GPU com o argumento `cudaMemcpyHostToDevice`. Na linha 27 é realizado o inverso com o argumento `cudaMemcpyDevicetoHost`. Do lado da GPU vemos que cada *thread* fica responsável pelo índice correspondente ao seu próprio ID, este ID pode ser obtido pelo comando `threadIdx.x` utilizado na linha 5. Cada *thread* salva o resultado de seu cálculo em seu respectivo índice c (linha 6). Ou seja, a *thread* de ID i soma os valores de $a[i]$ e $b[i]$ e salva em $c[i]$. Como nesse programa os valores são fixos, a saída sempre será $c[i] = a[i] + b[i]$. Nas linhas de 32 a 34 é feita a liberação de memória alocado no `cudaMalloc`. A chamada do *kernel* na linha 25 utiliza apenas um Bloco e N threads, onde N é definido como 32, dessa forma temos a execução de uma *warp* no lado da GPU, o que indica que todas as *threads* deste programa estarão sendo executadas em um mesmo MP.

Algoritmo 2.1: Programa exemplo em CUDA.

```
1  #include <fstream>
2  #define N 32
3  __global__ void add( int a[], int b[], int c[] )
4  {
5     int tid = threadIdx.x;
6     c[tid] = a[tid] + b[tid];
7  }
8  int main( void )
9  {
10    int i,a[N],b[N],c[N];
11    int *dev_a,*dev_b,*dev_c;
12    for(i=0;i<N;i++)
13    {
14       //a[0]=0 a[1]=1 a[i]=b[i]
15       a[i]=i;
16       b[i]=i;
17    }
18    cudaMalloc( (void**)&dev_a, N * sizeof(int) );
19    cudaMalloc( (void**)&dev_b, N * sizeof(int) );
20    cudaMalloc( (void**)&dev_c, N * sizeof(int) );
21
22    cudaMemcpy( dev_a, a, N * sizeof(int), cudaMemcpyHostToDevice );
23    cudaMemcpy( dev_b, b, N * sizeof(int), cudaMemcpyHostToDevice );
24
25    add<<<1,N>>>( dev_a, dev_b, dev_c );
26
27    cudaMemcpy( &c,  dev_c, N * sizeof(int), cudaMemcpyDeviceToHost );
28
29    for(i=0;i<N;i++)
30       printf( "c[%d]=%d\n",i,c[i]);
31
32    cudaFree( dev_a );
33    cudaFree( dev_b );
34    cudaFree( dev_c );
35    return 0;
36 }
```

2.3 Desempenho

Todas as placas Nvidia a partir da arquitetura Kepler tem uma funcionalidade que permite que *threads* acessem os registradores de outras *threads* sem utilizar a memória compartilhada desde que estas estejam em uma mesma *warp*, as instruções que permitem essa funcionalidade são chamadas de instruções *shuffle* [22].

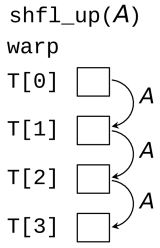
Um programa em GPU é executado por um conjunto de *threads* em *Single Instruction Multiple Data* (SIMD) na taxonomia Flynn [27]. Isto é, *threads* rodam a mesma operação utilizando dados diferentes. GPUs com CC 3.0 ou mais, permitem o uso de instruções *shuffle*, essas instruções permitem que *threads* de uma mesma *warp* acessem dados de outra thread de forma direta. Uma instrução *shuffle* tem menos latência que um acesso a memória global ou compartilhada. Elas são um bom artifício para aplicações que necessitam de troca rápida de dados entre *threads* [22].

Para o retorno destas instruções ser válido todas as *threads* precisam estar ativas, isto é, todas *threads* devem estar executando o comando *shuffle* no mesmo ciclo, caso contrário o retorno da instrução será indefinido, quando executada corretamente retorna um número de 32 bits correspondente ao valor do registrador de outra *thread*.

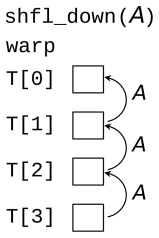
A instrução `shfl_up(`*var*`,1)` permite transferir o dado armazenado no registrador local *var* de uma thread cujo ID é imediatamente inferior da *thread* que chama a instrução. A Figura 2.3a mostra um exemplo da função `shfl_up()`. Nela, quatro *threads* T[0],T[1],T[2],T[3] usam a operação `shfl_up(`A`)` para transferir o conteúdo de seu registrador local A da *thread* T[$i-1$] a T[i], ($1 \leq i \leq w-1$) e com $w = 4$. Note que essa operação não necessita de acesso as memórias e a variável A é transferida entre as *threads* de uma mesma warp. Note também que a *thread* com menor ID (T[0]) não pega nenhum valor novo na operação `shfl_up()`. A operação `shfl_up()` retorna o valor do próprio registrador local A. A operação `shfl_down` mostrada na Figura 2.3b, funciona de uma forma similar, mas os valores são transferidos para as *threads* imediatamente superiores. Neste caso a *thread* com o maior ID (T[3]) não recebe valores de outras *threads*. A Figura 2.3c funciona de forma similar as anteriores, porém utiliza a memória compartilhada, Nesse caso são necessárias $w-1$ operações de leitura e $w-1$ operações de escrita.

A Figura 2.3a pode ser representada pelo Algoritmo 2.2 e a Figura 2.3c é representada pelo Algoritmo 2.3, note que No primeiro algoritmo é utilizado o comando `shfl_up()` na linha 7, enquanto no segundo é utilizado a memória compartilhada na linha 10.

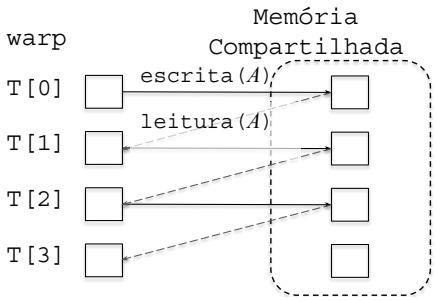
Podemos ver dois algoritmos que funcionam de forma similar, o Algoritmo 2.2 que utiliza comandos *shuffle* e o Algoritmo 2.3 que usa memória compartilhada. Caso o tamanho de A seja o tamanho de uma *warp* os dois programas terão as mesmas saídas para qualquer entrada. O programa que não utiliza comandos *shuffle* acaba tendo um uso



(a) Transferindo dados via `shfl_up()`



(b) Transferindo dados via `shfl_down()`



(c) alternativa usando utilizando memória compartilhada

Figura 2.3: Transferência de dados.

maior da memória compartilhada. Como essa memória tem tamanho limitado seu uso pode impedir armazenar outros dados.

Algoritmo 2.2: Código utilizando comando *shuffle*.

```
1 #include <fstream>
2 #define W 32 /*tamanho da warp*/
3 ___global___ void shift(int A[])
4 {
5     int tid=threadIdx.x;
6     /*comando retorna valor de A da thread de id inferior.*/
7     A=___shfl_up(A);
8     printf("T%d=%d",tid,A);
9 }
```

2.3.1 Métodos de Acesso a Memória Compartilhada

Nessa seção abordaremos a utilização da memória compartilhada e como podemos otimizar esse acesso. É necessário ter atenção para se fazer um programa eficiente quando se trata de GPUs. Okamoto *et al.* [28] mostraram a quantidade de ciclos que a GPU demora para vários casos de escrita e leitura de dados em memória compartilhada.

Podemos dizer que as principais memórias em CUDA são a memória compartilhada e a memória global. Uma GPU reúne vários SMs. Quando um programa é executado um

Algoritmo 2.3: Código utilizando memória compartilhada.

```
1  #include <fstream>
2  #define W 32 /*tamanho da warp*/
3  ___global___ void shift(int A[])
4  {
5      ___shared___ aux[W];
6      int tid=threadIdx.x;
7
8      aux[tid]=A;
9      if(tid!=0)
10         A=aux[tid-1];
11     printf("T%d=%d",tid,A);
12 }
```

bloco de *threads* é associado a um SM, cada SM tem sua própria memória compartilhada que pode ser acessada por qualquer *thread* pertencente ao bloco que está sendo executado, ou seja, a abrangência da memória compartilhada é apenas das *threads* pertencentes a um mesmo bloco.

Um valor salvo em um endereço de memória i é armazenado no (i mod w)-ésimo banco de memória onde w é o número de banco de memórias. Neste caso assume que o número de *warps* utilizado é o mesmo que o de bancos de memória. A Figura 2.4 mostra 3 exemplos de *threads* de uma mesma *warp* acessando os bancos de memória de forma diferente. O trabalho define k como sendo o número máximo de *threads* que tentam acessar um mesmo banco de memória em um mesmo ciclo, ou seja, se no máximo 4 *threads* podem acessar um mesmo banco de memória ao mesmo tempo, dizemos que $k = 4$.

Um conflito ocorre quando duas ou mais *threads* tentam acessar um mesmo banco de memória no mesmo ciclo de tempo. Caso um conflito ocorra estes acessos são feitos em turnos. Note que conflitos devem ser evitados, pois podem aumentar consideravelmente o tempo de acesso à memória. Por exemplo, se duas *warps* tentam acessar um mesmo endereço de memória compartilhada ao mesmo tempo, há um conflito, fazendo com que uma *warp* acesse este conteúdo em um tempo t_0 e outra em t_1. A Figura 2.4 mostra exemplos em que ocorrem um número diferente de k conflitos em cada um dos exemplos, onde a Figura 2.4(a) mostra o acesso livre, ou seja o melhor cenário, e as Figuras 2.4(b) e 2.4(c) mostram 4 e 2 conflitos respectivamente.

O exemplo da Figura 2.4(a) seria o com menor latência pois todas *threads* acessariam os bancos de memória ao mesmo tempo, o exemplo da Figura 2.4(b) seria com maior latência, seria feito em 4 turnos e a Figura 2.4(c) seria feito em 2 turnos.

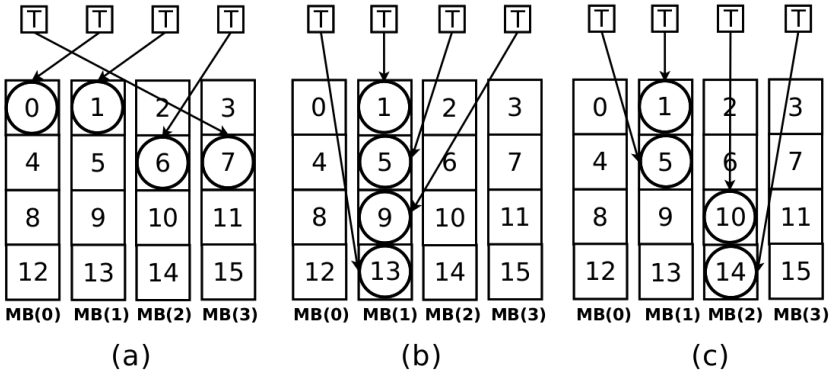


Figura 2.4: Exemplos de acesso com $w = 4$ feito em 1, 4 e 2 turnos respectivamente [28].

2.3.2 Tempo de Acesso

Okamoto *et al.* [28] apresenta um estudo sobre o número de ciclos necessários para realizar instruções de *load/store* em uma placa GTX780Ti. Para este fim, foi utilizado as instruções *inline PTX assembly* que permitem utilizar comandos *assembly* da GPU em meio ao código CUDA [29]. O Algoritmo 2.4 mostra um pedaço do código utilizado, a linha 4 garante que todas as *threads* estejam sincronizadas, as linhas 5 e 7 gravam o número de ciclos de *clock* naquele momento, a linha 6 representa as instruções de *load* ou *store*.

Algoritmo 2.4: Código utilizado para medir quantidade de ciclos [28].

```
1 asm volatile
2 (
3     ...
4     "bar.sync 0; \n\t"
5     "mov;u32 %0, %%clock; \n\t"
6     instruções load/store na memória compartilhada.
7     "mov;u32 %0, %%clock; \n\t"
8     ...
9 )
```

Os resultados foram coletados com base nos seguintes parâmetros: (i) o número de *warps*; (ii) o número k de conflitos; (iii) o número de *loads/stores* executados na memória compartilhada. Foram mostrados 1 ou 32 *warps*. Cada *warp* tem 32 *threads* logo k pode ser no máximo 32.

Foram mostrados apenas os resultados para instruções *load*. Entretanto os autores ressaltam que as instruções *store* obtiveram resultados similares. A Figura 2.5 mostra a quantidade de ciclos necessários para executar uma instrução *load* com 1 e 32 *warps*. O tempo de execução aumenta de acordo com o número de acessos conflitantes k, os resultados são lineares, e o número de ciclos de *clock* com $k = 32$ cresce mais de 33

vezes em relação a $k = 1$ para uma *warp*. O aumento em 33 vezes mostra que quando se tem conflito no acesso as *threads* devem esperar seu turno para acessar a memória compartilhada, tornando assim o acesso serializado. Ao contrário do esperado o acesso com 32 *warps* tem um aumento aproximado em 16 vezes, isso indica um certo paralelismo de execução destas *warps* [28].

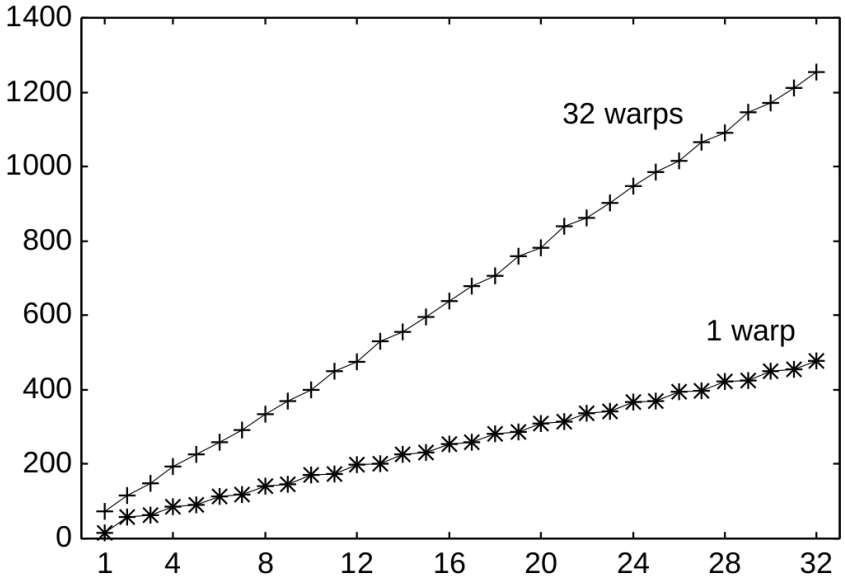


Figura 2.5: Quantidade de ciclos por k colisões [28].

A Figura 2.6 mostra a quantidade de ciclos necessários para executar um número variado de instruções para $k = 1$. Para uma *warp* quando executamos 32 instruções vemos o aumento na quantidade de ciclos em aproximadamente 31 vezes em comparação com a execução de uma instrução. Para 32 *warps* o aumento é de aproximadamente 14 vezes, indicando um certo paralelismo visto que a quantidade de ciclos necessários para a execução das 32 instruções das 32 *warps* foi apenas 2,23 vezes maior que as 32 instruções executadas por uma *warp* [28]. A Figura 2.7 é similar a Figura 2.6, porém, utiliza $k = 32$ colisões pra cada instrução. Para uma *warp* o número de ciclos para 32 instruções cresceu aproximadamente 32 vezes em relação a uma instrução, para 32 *warps* este número cresceu aproximadamente 26 vezes [28], o que mostra como o acesso conflitante.

Os resultados mostram que para uma *warp* o tempo cresce quase linearmente em relação a k, o que indica uma serialização do acesso à memória compartilhada, porém, quando se utilizam várias *warps* o resultado indica uma execução concorrente. Podemos ver isso na Figura 2.6 onde o número de ciclos para 32 instruções cresceu aproximadamente 14 vezes em relação a uma instrução.

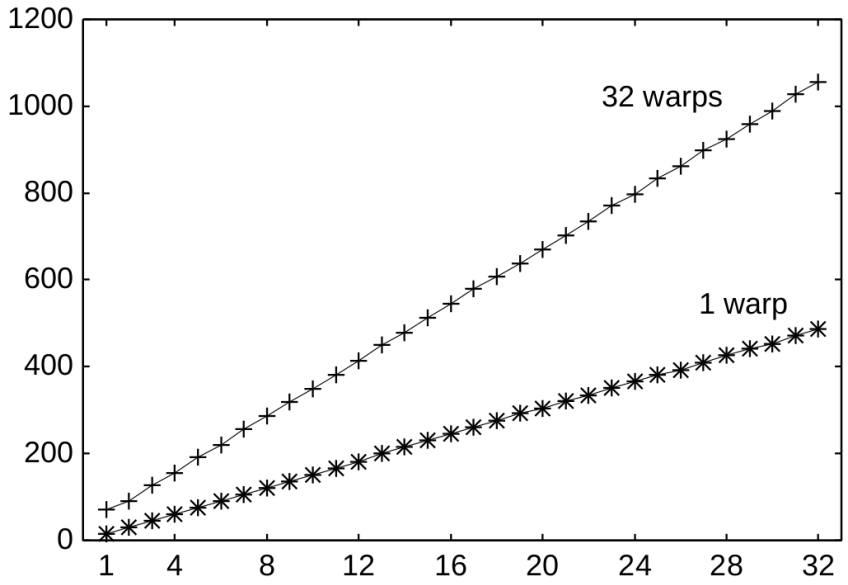


Figura 2.6: Quantidade de ciclos variando o número de instruções e com $k = 1$ [28].

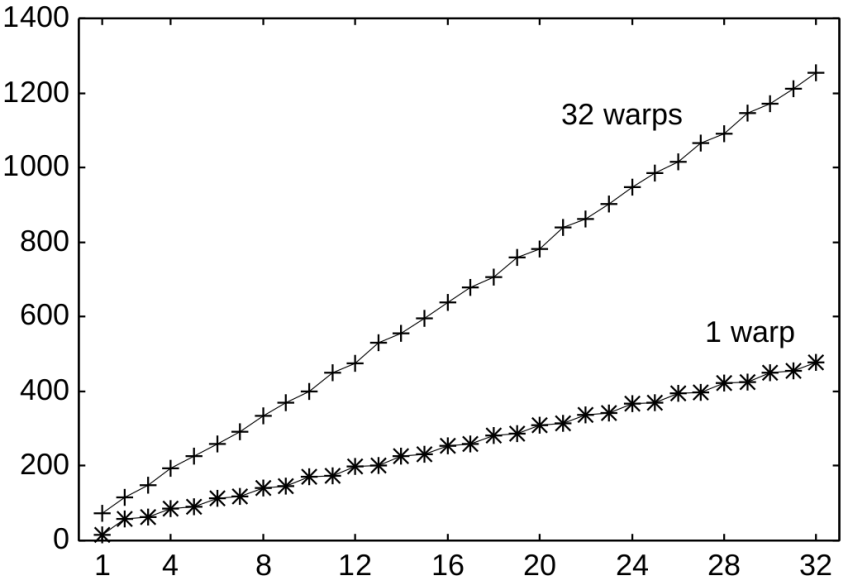


Figura 2.7: Quantidade de ciclos variando o número de instruções e com $k = 32$ [28].

2.3.3 Ocupação de GPU

Ocupação é definida como a taxa de *warps* ativas em um MP em relação ao número de *warps* ativas suportado. A capacidade do dispositivo, e as opções de compilação podem limitar esse número.

Como exemplo utilizamos uma placa GTX 960 [30], que tem uma CC de 5.2. O número máximo de *warps* por MP nesta placa é 64. Porém, o número de *warps* sendo executadas esta sujeita à variação por:

1. **Número de blocos por MP**: A placa GTX 960 tem um limite de 32 blocos por MP. A Figura 2.8 mostra a utilização de *warps* em um MP variando o número de *threads* por bloco. Com 32 *threads* (1 *warp*) utilizamos apenas metade da capacidade do MP, o que limitaria sua ocupação em 50%. Nos casos de $64, 128, 256, 512$ e 1024 *threads* por bloco, a ocupação seria de 100%. Por exemplo, 1024 *threads* (32 *warps*) por bloco, a ocupação da GPU chega a 100% ocupando 2 blocos por MP.

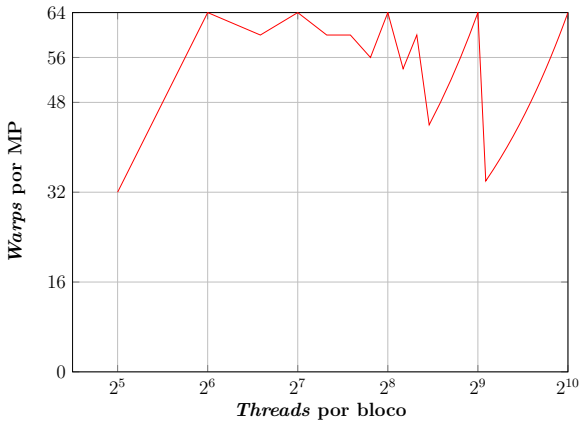


Figura 2.8: *Warps* por MP com um número crescente de *threads* por bloco.

2. **Registradores por MP**: Cada MP possui um limite de registradores, no caso de dispositivos com CC 5.2 são $64K$ registradores de 32 bits cada. Com a execução de 32 blocos por MP, teríamos $\frac{64k}{32} = 2k$ registradores por bloco. Com 64 *threads* por bloco, cada *thread* teria $\frac{2K}{64} = 32$ registradores. Como mostrado na Figura 2.9, dobrando o uso do bloco (por exemplo, aumentado para 128 *threads*) e mantendo o limite de 32 registradores, o MP atingiria seu limite com 16 blocos.

3. **Memória Compartilhada por MP**: A GTX 960 tem 96KB de memória compartilhada por MP. Com 32 blocos, cada bloco teria $\frac{96KB}{32} = 3KB$ de memória compartilhada. Conforme a Figura 2.9, se aumentarmos o número de *threads* o número máximo de blocos é reduzido e consequentemente, aumenta-se a quantidade de memória compartilhada disponível por bloco.

2.3.4 Nvidia Profiler

A Nvidia disponibiliza a ferramenta chamada de Profiler [31], que permite ver atividades da GPU e da CPU. Dentro destas atividades é possível ver o tempo de execução em cada *kernel*, atributos como o tempo de transferência de dados, a ocupação da GPU e o número de acessos conflitantes.

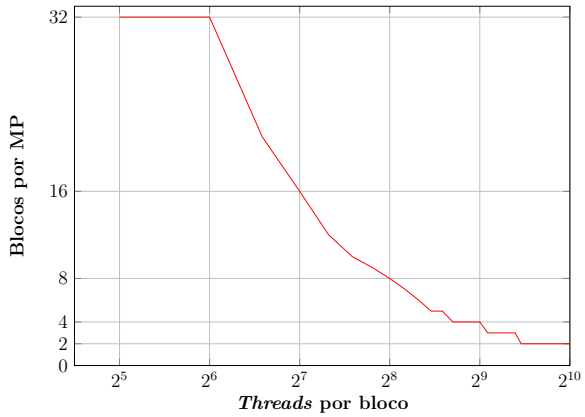


Figura 2.9: Blocos por MP com um número crescente de *threads* por bloco.

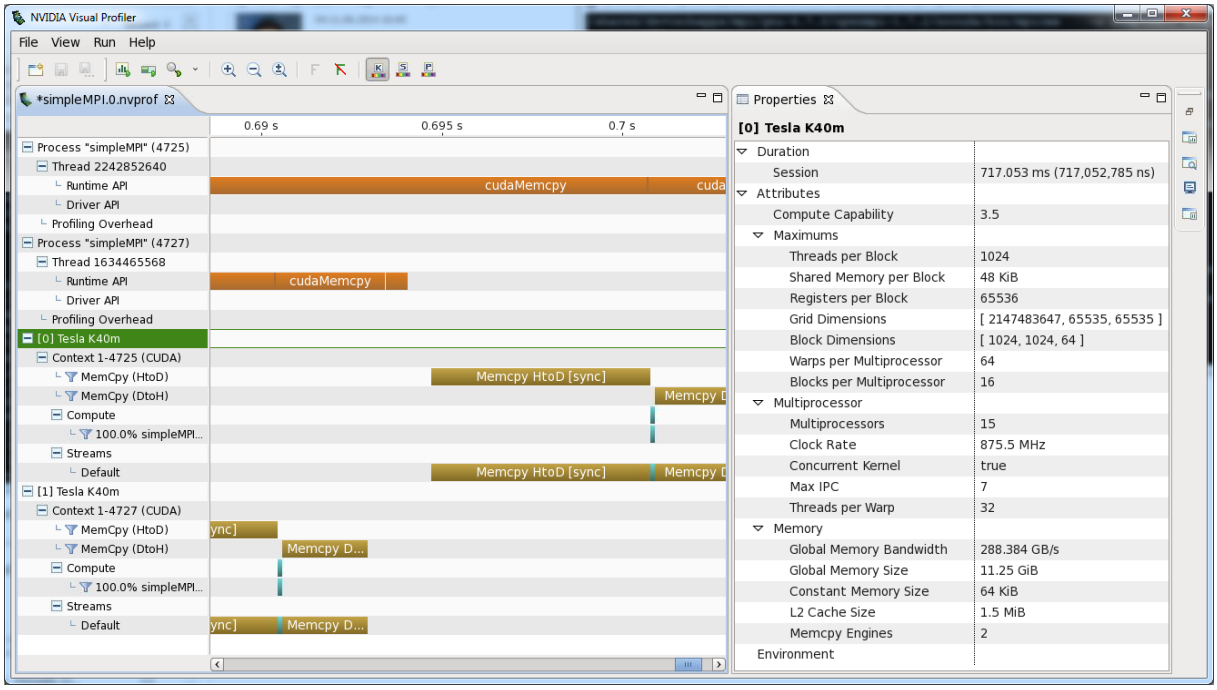


Figura 2.10: Tela exemplo do Nvidia Visual Profiler [22].

A Figura 2.10 mostra um exemplo do Profiler em modo gráfico. Nele podemos visualizar em forma de linha do tempo toda a execução de um programa CUDA. A ferramenta utilizada pelo Profiler é o `nvprof`, que tem uma série de eventos e métricas que podem ser extraídas da execução de um binário, por exemplo, com o evento `shared_load` temos o total de instruções loads são executados por *warp* na memória compartilhada, da mesma forma podemos ver o conflito de escrita em memória compartilhada com o evento `shared_st_bank_conflict`. Na parte de métricas conseguimos checar a ocupação da GPU com a métrica `achieved_occupancy`, ou a velocidade de acessos das memórias, por exemplo, `local_load_throughput` retorna à velocidade de leitura da memória local.

Dessa forma o Profiler é uma ferramenta para análise e pode indicar possíveis gargalos em um programa. Também é útil para evidenciar diferença de desempenho entre algoritmos.

Neste capítulo vimos um pouco da história e os principais componentes de uma GPU Nvidia, o funcionamento de um programa em CUDA, a utilização de instruções *shuffle* e o uso da memória compartilhada, como garantir que se utilize uma boa ocupação da placa e apresentamos a ferramenta de Profiler. No próximo capítulo veremos algoritmos conhecidos na literatura e que serviram como referência para nosso trabalho.

Capítulo 3

Casamento de Padrões

O problema de casamento de padrões tem várias aplicações práticas e, por consequência, tem sido bastante estudado [32] [33]. Este problema visa encontrar todas as ocorrências de um certo padrão em um texto. Um exemplo seria um editor de texto que precisaria buscar uma palavra, outro exemplo seria encontrar sequencias de DNA em um código genético [12]. Este capítulo apresenta uma breve revisão das técnicas de casamento de padrões e suas variações.

O casamento de padrões pode ser exato ou aproximado [34]. Este capítulo aborda uma visão geral sobre as técnicas de padrão exato e aproximado.

3.1 Casamento de Padrões Exato

O problema de reconhecimento de padrões exato pode ser definido como um texto $T[1..n]$ de tamanho n e um padrão $P[1..m]$ de tamanho m onde $m \leq n$ [35]. Assumimos que todos elementos em P e T são caracteres definidos em um alfabeto finito Σ. Podemos ter um alfabeto de DNA, por exemplo $\Sigma = \{A, T, C, G\}$ ou de texto $\Sigma = \{a, b, \ldots, z\}$. Chamamos esses conjuntos de caracteres de P ou T de cadeias de caracteres [35].

Dizemos que um padrão P tem um casamento com T se após um deslocamento s todos os caracteres coincidirem. O deslocamento é valido se $0 \leq s \leq n-m$ e $T[s+1, \ldots s+m] = P[1 \ldots m]$, ou seja o padrão deve ser igual ao texto em todos seus caracteres com um deslocamento válido de acordo com o tamanho do texto e do padrão. Um exemplo de casamento de padrões está na Figura 3.1, onde temos um deslocamento $s = 4$ para encontrar um casamento válido entre P e T.

Dizemos então que o problema de reconhecimento de padrões em texto consiste em encontrar ocorrências de uma cadeia de caracteres conhecida como padrão em um determinado texto. Existem diversos algoritmos na literatura, entre eles o Rabin-Karp (RK) [32] e o Knuth-Morris-Pratt (KMP) [33]. Vamos mostrar aqui alguns deles e mostrar seu fun-

cionamento e complexidade. A Tabela 3.1 mostra as complexidades de pré-processamento e do algoritmo de busca para cada um dos algoritmos que vamos citar nas seções seguintes.

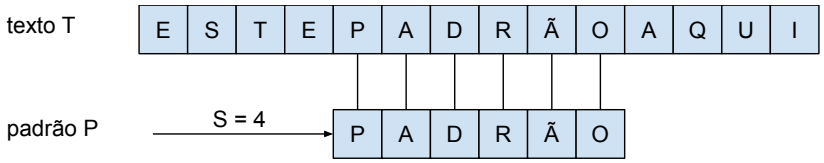


Figura 3.1: Exemplo de padrão encontrado em um texto.

Tabela 3.1: Complexidade dos algoritmos de Casamento de Padrões [35].

Algoritmo	Tempo de Pré-processamento	Tempo de *Match*
Ingênuo	0	$O((n-m+1)m)$
Rabin-Karp	$\Theta(m)$	$O((n-m+1)m)$
Autômato Finito	$O(m\|\Sigma\|)$	$\Theta(n)$
KMP	$\Theta(m)$	$\Theta(n)$

3.1.1 Autômato Finito

Um algoritmo pode também utilizar um autômato finito que procura no texto T todas as ocorrências do padrão P [35]. Essa técnica é muito eficiente visto que eles somente visitam cada caractere em um texto uma vez. Dessa forma a complexidade de tempo dessa técnica é $\Theta(n)$. O pré-processamento pode ser um problema já que varia de acordo com o alfabeto Σ.

Para construir precisamos definir a tupla$(Q, q_0, A, \Sigma, \delta)$, onde:

- Q é um número finito de estados;
- $q_0 \in Q$ é o estado inicial;
- $A \subseteq Q$ são os estados aceitos;
- Σ é um alfabeto finito;
- δ é uma função de $Q \times \Sigma$ em Q, chamada **função de transição** de M.

Se o autômato começa no estado q e lê o caractere a, este vai para o estado $delta(q, a)$. Isso quer dizer que o estado q é membro de A, e a máquina M aceitou os caracteres lidos até agora. Uma entrada não aceita é dada como rejeitada. A Figura 3.2 mostra um exemplo de estados possíveis, vemos, por exemplo que a transição do estado 0 para 1 precisaria da entrada a.

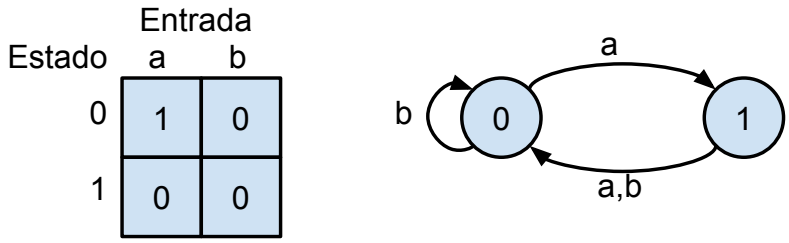


Figura 3.2: Um simples autômato com estados $Q = [0, 1]$ [35].

Algoritmo 3.1: Algoritmo de autômato finito [35]

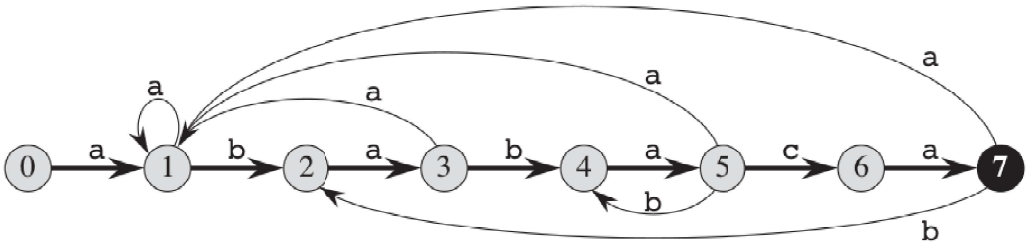
Entrada: T,δ,m
// T *string* do texto
// δ função de transição
// m tamanho do padrão
1 n $\leftarrow |T|$
2 q $\leftarrow 0$
3 **for** $i \leftarrow 1$ *até* n **do**
4 $\quad q \leftarrow \delta(q, T[i])$ **if** $q = m$ **then**
$\quad\quad$ // Imprime mensagem concatenando o resultado $i - m$
5 $\quad\quad$ imprima "Padrão ocorre com deslocamento = " $+ i - m$
6 $\quad$ **end**
7 **end**

O Algoritmo 3.1 apresenta uma possível implementação para os estados da Figura 3.2. A Figura 3.3 mostra um exemplo de utilização desse método, onde temos o padrão `ababaca` a ser encontrado no texto, o estado 0 como o estado inicial e o estado 7 como o estado aceito, ou seja, o padrão será encontrado. Uma aresta de um estado i para um estado j com nome a representa $\delta(i, a) = j$. Podemos ver na tabela no lado esquerdo as correspondentes funções de transição δ de acordo com o caractere do padrão P. Na representação do lado direito vemos que o padrão é encontrado no texto apenas uma vez terminando na posição 9 [35].

3.1.2 Algoritmo de Rabin-Karp

Rabin e Karp [32] propuseram um algoritmo que generaliza outros algoritmos para problemas relacionados , por exemplo, encontrar o padrão em duas dimensões. O pior caso tem complexidade de tempo $O((n-m+1)m)$, porém, o algoritmo utiliza um pré-processamento que leva $O(m)$.

O algoritmo de Rabin-Karp procura transformar sequências de caracteres em números, por exemplo. Digamos que o alfabeto seja $\Sigma = \{0, 1, 2, 3, \ldots, 9\}$ e vamos considerar cada



estado	Entrada a	b	c	P
0	1	0	0	a
1	1	2	0	b
2	3	0	0	a
3	1	4	0	b
4	5	0	0	a
5	1	4	6	c
6	7	0	0	a
7	1	2	0	

i	—	1	2	3	4	5	6	7	8	9	10	11
$T[i]$	—	a	b	a	b	a	b	a	c	a	b	a
Estado $\phi(T_i)$	0	1	2	3	4	5	4	5	6	7	2	3

Figura 3.3: Exemplo utilizando autômato finito [35]

um destes caracteres como dígito decimal. Dessa forma podemos ver uma sequência de caracteres como um número. Por exemplo, temos a sequência de caracteres 12345, o algoritmo transforma isto em um número inteiro 12345. Neste caso podemos representar uma palavra de k caracteres como um número decimal de k dígitos.

O pré-processamento do algoritmo seria encontrar o valor p correspondente ao padrão P, abaixo vemos como é calculado o valor do padrão P

$$p = P[m] + 10(P[m-1] + 10(P[m-2] + 10(P[m-3] + \cdots + 10(P[2] + 10P[1])))).$$

De forma similar ao valor calculado acima, calculamos t_0 que corresponde aos m primeiros caracteres do texto T e levaria tempo de $\theta(m)$, e o valor t_1 teria que desconsiderar o primeiro caractere e utilizar o caractere $m+1$. Dessa forma os valores seguintes demorariam um tempo de $\theta(n-m)$ utilizando a seguinte fórmula

$$t_{s+1} = 10(t_s - 10^{m-1}T[s+1]) + T[s+m+1].$$

O problema é que se considerarmos m como tamanho grande seria inverídico considerar

a comparação do número p com t_i constante, pois há um limite de tamanho que uma palavrade computador consegue armazenas, por exemplo, se a palavra é de 32 bits, conseguimos armazenar números de 0 a $2^{32} - 1$ Com a definição de um número primo q podemos calcular um *hash* simplesmente pegando o módulo do que seria o resultado da cadeia de caracteres.

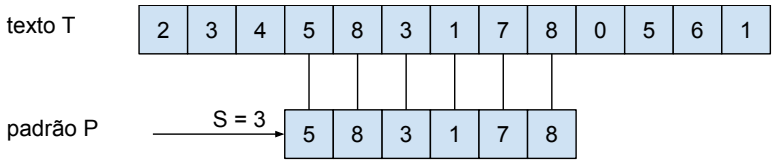


Figura 3.4: Exemplo do algoritmo de Rabin-Karp com $S = 3$

Com estas definições o Algoritmo 3.2 é utilizado. Na linha 3 o h é inicializado com o valor máximo possível de uma janela m, as linhas de 7 a 8 calculam p como o valor correspondente a $P[1 \ldots m] \bmod q$ e t_0 como valor correspondente de $T[1 \ldots m] \bmod q$. No laço for da linha 10 são feitos todos os deslocamentos possíveis, na linha 11 verificamos se o *hash* do texto é igual ao *hash* do padrão, como característica, um *hash* igual não significa que os caracteres do texto e do padrão naquela posição são iguais, por isso não se pode ter um q pequeno. Na linha 12 uma vez que verificamos que o hash é igual verificarmos caractere a caractere se o padrão casa com o texto e a linha 16 faz o cálculo do novo valor t_s de acordo com o deslocamento de s [35].

Podemos ver um exemplo na Figura 3.4 onde as cadeias são transformadas em números e sempre é feita a comparação entre inteiros, economizando assim a um possível processamento de caractere a caractere.

3.1.3 Algoritmo Knuth-Morris-Pratt

O algoritmo de Knuth-Morris-Pratt (KMP) [33] é muito utilizado na literatura para o problema de encontrar padrões em um texto. O algoritmo consegue realizar o casamento em tempo linear $\Theta(n)$ e tem um pré-processamento com complexidade de tempo $\Theta(n)$ [35]. Este pré-processamento utiliza uma função auxiliar $\pi[1 \ldots m]$ ao invés de utilizar a função de transição σ.

O algoritmo funciona encontrando prefixos no padrão, esses prefixos são achados comparando o padrão com o próprio padrão. Por exemplo, temos o padrão $P = ababaca$ e um texto $T = bacbababababcbab$ se considerarmos o deslocamento $s = 4$ vamos ter um casamento parcial, apenas dos 5 primeiros caracteres, a partir destes 5 caracteres podemos inferir que a posição $s + 1(5)$ é inválida e não precisa ser checada, mas a posição $s + 2(6)$ pode ser válida, ou seja, o algoritmo sabe onde começar uma nova comparação. Dessa

Algoritmo 3.2: Algoritmo de Rabin-Karp [35].

```
   Entrada: P,T,d,q
   // P string do padrão
   // T string do texto
   // d tamanho do alfabeto (|Σ|)
   // q número primo
 1 n ← |T|
 2 m ← |p|
 3 h ← d^{m−1} mod q
 4 p ← 0
 5 t_0 ← 0
   // Pré-processamento
 6 for i ← 1 até m do
 7 |   p ← (dp + P[i]) mod q
 8 |   t_0 ← (dt_0 + T[i]) mod q
 9 end
   // Matching
10 for s ← 0 até n − m do
11 |   if p = t_s then
12 |   |   if P[1..m] = T[s + 1..s + m] then
   |   |   |   // Imprime mensagem concatenando o resultado s
13 |   |   |   imprima "Padrão ocorre com deslocamento = " + s
14 |   |   end
15 |   |   if s < n − m then
16 |   |   |   t_{s+1} ← (d(t_s − T[s + 1]h) + T[s + m + 1]) mod q
17 |   |   end
18 |   end
19 end
```

forma podemos analisar o padrão de forma a não precisar verificar todos os caracteres para cada posição, o Algoritmo 3.3 mostra o pré-processamento retornando à função auxiliar π, na Tabela 3.2 podemos ver um resultado para o padrão $P = ababaca$, note que a cada ocorrência de um novo caractere o valor no vetor naquela posição π é zerado. Com esses valores contidos em π, já é possível encontrar as ocorrências do padrão P no texto T na complexidade de tempo $\Theta(n)$. O algoritmo 3.4 mostra o funcionamento do KMP, utilizando o exemplo anterior com um padrão $P = ababaca$ e um texto $T = bacbababaabbab$, quando o deslocamento s fosse 4 e estivéssemos comparando o caractere c do padrão com o texto, o código entraria no *while* setando $q = 3$ e continuando a comparação com este novo deslocamento.

Algoritmo 3.3: Pré-processamento do KMP [35]

```
   Entrada: P
   // P string do padrão
 1 m ← |P|
 2 k ← 0
 3 π[1] ← 0
 4 for q ← 2 até m do
 5 |   while k > 0 and P[k + 1] ≠ P[q] do
 6 |   |   k ← k + 1
 7 |   end
 8 |   if P[k + 1] = P[q] then
 9 |   |   k ← k + 1
10 |   |   π[q] ← k
11 |   end
12 end
13 Retorne π
```

Tabela 3.2: Exemplo de π para o pré-processamento do KMP.

P	a	b	a	b	a	c	a
q	1	2	3	4	5	6	7
π	0	0	1	2	3	0	1

3.2 Casamento de Padrões Aproximado

Esta classe de algoritmos tem como foco encontrar cadeias de caracteres semelhantes. Técnicas para acelerar estas soluções vêm sendo exploradas, inclusive por algoritmos paralelos [2] [12]. Esta classe de algoritmos tem por objetivo encontrar onde duas entradas são mais similares. Nesta seção iremos explorar o casamento de padrões aproximado sem peso, e será falado sobre distância de edição com peso.

3.2.1 Distância de Edição

Suponha duas cadeias de caracteres X e Y de tamanho m e $n(m \leq n)$ respectivamente, se deseja transforma a *string* Y em X, para isso podem ser usados três operações: inserção, deleção e alteração de caracteres. A Distância de Edição (DE) $DE(X, Y)$ é a quantidade mínima de operações necessárias para transformar Y em X. O *Approximate String Matching* (ASM) é uma versão mais flexível já que este retorna o menor valor de DE para qualquer sequência de caracteres de Y. O ASM é definido na Equação 3.1.

$$ASM(X, Y) = min\{DE(X, Y') \mid Y' \text{ é } uma\ substring\ de\ Y\}. \tag{3.1}$$

Algoritmo 3.4: Algoritmo KMP [35].

```
   Entrada: T,P
   // P string do padrão
   // T string do texto
 1 n ← |T|
 2 m ← |P|
   // pré-processamento
 3 π ← pré-processamento(P)
 4 q ← 0
 5 for i ← 1 até n do
 6 |   while q > 0 AND P[q + 1] ≠ T[i] do
 7 |   |   q ← π[q]
 8 |   end
 9 |   if P[q + 1] = T[i] then
10 |   |   q ← q + 1
11 |   end
12 |   if q = m then
   |   |   // Imprime mensagem concatenando o resultado i − m
13 |   |   imprima "Padrão ocorre com deslocamento = " + i − m
   |   |   // procura próxima ocorrência
14 |   |   q ← π[q]
15 |   end
16 end
```

Quanto mais similar uma cadeia Y estiver de X menor será o valor de $ASM(X,Y)$, o valor máximo dessa função seria m por se tratar do tamanho de Y, e o valor mínimo seria 0 quando uma cadeia de Y fosse igual a X. Mais detalhes podem ser encontrados em [36]. Podemos calcular este ASM computacionalmente utilizando uma matriz c de tamanho $(m+1) \times (n+1)$, onde cada célula da matriz seria usada para armazenar a distância de edição de sequências de caracteres de X e Y. Dessa forma, os valores armazenados em $c[i][j](0 \leq i \leq m, 0 \leq j \leq n)$ seriam calculados segundo a Equação 3.2.

$$c[i][j] = \min_{1 \leq j' \leq j} DE(x_1, x_2 \cdots x_i, y_{j'} y_{j'+1} y_j). \tag{3.2}$$

Uma vez que essa matriz está preenchida podemos retirar o valor do ASM como o menor valor da última linha, já que a última linha representa a cadeia de caracteres completa de X com cada uma das sequências de caracteres de y, chegamos assim na Equação 3.3.

$$ASM(X,Y) = \min_{0 \leq j \leq n} c[m][j]. \tag{3.3}$$

.

Algoritmo 3.5: Algoritmo ASM sequencial.

```
1  for i ← 0 até m do
2  |  c[i][0] ← i
3  end
4  for j ← 1 até n do
5  |  c[0][j] ← 0
6  end
7  for i = 1 até m do
8  |  for j = 1 até n do
9  |  |  c[i][j] ← min(c[i][j − 1] + 1, c[i − 1][j] + 1, c[i − 1][j − 1] + (x_i ≠ y_i))
10 |  end
11 end
12 Imprima min{c[m][j]|0 ≤ j ≤ n}
```

Uma forma de calcular a DE usando valores já computados é utilizando 3 células vizinhas(da esquerda, de cima, diagonal entre estas), deve-se considerar se o atual caractere x_i é igual a y_j pois isso aumenta a distância de edição, podemos ver o cálculo na Equação 3.4.

$$c_{i,j} = min \begin{cases} c_{i-1,j} + 1 \\ c_{i,j-1} + 1 \\ c_{i-1,j-1} + (x_i \neq y_j) \end{cases} \tag{3.4}$$

	Y	a	a	a	b	b	b
X	0	0	0	0	0	0	0
a	1	0	0	0	1	1	1
a	2	1	0	0	1	2	2
b	3	2	1	1	0	1	2
a	4	3	2	1	1	1	2
a	5	4	3	2	2	2	2

Figura 3.5: Exemplo de matriz c resultante.

onde $(x_i \neq y_j)$ retorna um valor binário tal que $x_i \neq y_j \rightarrow 0$ e $x_i = y_j \rightarrow 1$, dessa forma calcula-se o valor ASM(X,Y) de acordo com o Algoritmo 3.5. Na Figura 3.5 temos um possível exemplo da matriz c, onde a saída do algoritmo seria 2, pois este é o menor valor da última linha, ou seja, é onde se encontra a *substring* de Y que mais se assemelha a X.

3.2.2 Distância de Edição com Pesos

Na Seção 3.2.1 vimos como é realizado o cálculo do ASM, onde cada operação de um caractere, seja inserção, alteração ou deleção aumentava em 1 a DE. Porém em algumas aplicações a substituição de caracteres pode ter pesos diferentes dependendo dos caracteres a serem substituídos. Por exemplo, se uma aplicação quer encontrar palavras digitadas e quer considerar possíveis erros de digitação em teclado, a substituição de um caractere a por f vai ter um peso δ superior ao caractere s que fica ao lado do original no leiaute QWERTY [37].

O algoritmo de DE com pesos é similar ao Algoritmo 3.5, porém ao invés da linha 8 seria usado a Equação 3.5.

$$c_{i,j} = min \begin{cases} c_{i-1,j} + excluir(x_i) \\ c_{i,j-1} + inserir(y_j) \\ c_{i-1,j-1} + susbtituir(x_i, y_j). \end{cases} \tag{3.5}$$

As 3 funções acima retornariam valores diferentes dependendo dos caracteres (x_i e y_j). Por exemplo, a função $excluir(x_i)$ poderia retornar 2, a de função $inserir(y_j)$ retornar 1, a função de $susbtituir(x_i, y_j)$ poderia retornar 1 se os caracteres x_i e y_j forem ambos vogais ou ambos consoantes e retornar 2 caso um seja vogal e outro consoante.

3.2.3 ASM em GPU: Uma Implementação Ótima

Como trabalho relacionado temos um algoritmo de ASM apresentado por Nakano *et al.* [2] que vamos chamar de O-ASM, o trabalho mostra uma técnica ótima em tempo de execução para encontrar o ASM de duas cadeias de caracteres. O trabalho se utiliza dos modelos de memória descritos em [38] [39] e análises matemáticas para provar que o algoritmo é ótimo em tempo de execução.

Índices de Acesso à Memória

Um número p de *threads* são particionadas em $\frac{p}{w}$ *warps*, onde w representa o número de *threads* em uma *warp*. *Warps* podem revezar seu acesso a memória em turnos, já as *threads* dentro das *warps* podem tentar acessar bancos de memória simultaneamente. Quando essas *threads* tentam acessar um mesmo banco é dito que houve um acesso simultâneo. Como mostrado na Seção 2.3.1 as *threads* acabam acessando este banco em turnos, caso contrário todas conseguem acessar as respectivas posições de memórias em um mesmo ciclo. A latência de acesso da memória compartilhada é pequena se comparada com a da memória global, vamos chamar essas latências de l e L respectivamente.

Pela Equação 3.4 vemos que para calcular cada uma das célula são necessárias 3 células vizinhas, dessa forma pode-se reduzir o armazenamento necessário. Para isso precisamos de 3 diagonais, onde por exemplo a diagonal k representaria a diagonal em que se encontra a célula calculada. A diagonal $k-1$ é a dos vizinhos de cima e da esquerda. A última $k-2$ é a diagonal do último vizinho. Desta forma só seria necessário utilizar uma matriz de tamanho $3 \times (m+1)$.

Vamos considerar e uma matriz de tamanho $3 \times (m+1)$ onde cada valor $c[i][j] (0 \leq i \leq m, 0 \leq j \leq n)$ é armazenado em $e[j \ mod \ 3][i]$. Assim podemos calcular o ASM utilizando o Algoritmo 3.6. A Figura 3.6 mostra o uso da matriz e onde cada k representa uma diagonal e as células em evidência fazem parte da matriz e de tamanho $3 \times (m+1)$.

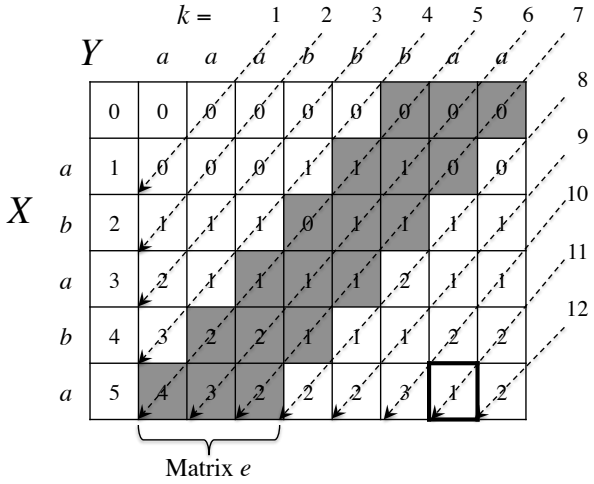


Figura 3.6: Algoritmo ASM paralelo com armazenamento reduzido.

Algoritmo O-ASM

Para realizar o cálculo do ASM em uma GPU, devemos paralelizar a matriz em diagonais e reduzir seu armazenamento a uma matriz e, também é necessário dividir as tarefas de cada *thread*, considerando as comunicações possíveis. Para um algoritmo em GPU devemos separar a execução em d blocos ou d partes. Como já vimos anteriormente não temos garantia da ordem de execução e nem a sincronia destes blocos, logo todas essas partes devem ser independentes e não devem precisar de qualquer comunicação.

A distância de edição é o número mínimo de operações necessárias para transformar Y em X. Podemos dizer que $n-m \leq DE(X,Y) \leq n$. O número de operações sempre vai ser maior que a diferença de tamanho entre X e Y pois de qualquer maneira será necessário no mínimo $n-m$ caracteres para preencher essa diferença. O número de operações não pode ultrapassar n visto que n operações poderia transformar qualquer sequência em X. De outra forma pode se dizer que se $n > 2m$ então $DE(X,Y) > m$.

Algoritmo 3.6: Algoritmo ASM paralelo para GPU [2].

```
1  resultado ← m
2  e[0][0] ← 0
3  e[0][1] ← 1
4  e[1][0] ← 0
5  for k ← 1 até n+m-1 do
6  |  j ← k − threadID + 1
7  |  if i = 0 then
8  |  |  e[j mod 3][i] ← 0
9  |  end
10 |  else if j = 0 then
11 |  |  e[j mod 3][i] ← i
12 |  end
13 |  else if 1 ≤ j ≤ n then
14 |  |  e[j mod 3][i] ← min(e[i][j − 1 mod 3] + 1, e[i − 1][j mod 3] + 1, e[i −
   |  |    1][j − 1 mod 3] + (x[i] ≠ y[j]))
15 |  end
16 end
```

A definição do ASM pode ser vista na Equação 3.1, se conseguirmos separar Y em d cadeias de caracteres podemos reescrever resultando na Equação abaixo.

$$ASM(X,Y) = min\{DE(X,Y_i) \mid 0 \leq i \leq d-1\}. \tag{3.6}$$

Calculando o $ASM(X,Y_i)$ para todo i $(0 \leq i \leq d-1)$ em paralelo, encontraríamos o valor de $ASM(X,Y)$. Devemos separar esse Y de forma que cada Y_i seja independente. Qualquer sequência de caracteres em Y que tenha mais de $2m$ caracteres pode ser ignorada visto que o resultado do algoritmo seria pelo menos m, qualquer sequência de caracteres em Y menor que m resultaria em um ASM de no máximo m.

Devemos garantir que cada parte Y_i tenha todas cadeias de caracteres de Y com no máximo $2m$ caracteres. Se simplesmente separássemos o Y em d partes de $\frac{n}{d}$ caracteres cada, deixaríamos de avaliar, por exemplo, uma cadeia de caracteres envolvendo os últimos caractere de Y_0 com os caracteres iniciais de Y_1.

O correto seria estender pelo menos $2m$ caracteres em cada parte Y_i, assim cada parte teria $\frac{n}{d}+2m$ caracteres, porém isso não se aplica ao Y_{d-1} já que não existem mais caracteres para estender. Como queremos uma divisão igualitária $s = \frac{n-2m}{d}$ é definido como a cadeia de caracteres Y sem os últimos $2m$ caracteres (vamos adicioná-los depois) divididos pelas d partes e cada uma dessas partes deve se estender $2m$ caracteres.

Dessa forma $Y_i = y_{is}y_{is+1}\cdots y_{(i+1)s+2m-1}$ para todo i $(0 \leq i \leq d-1)$, onde $s = \frac{n-2m}{d}$. Por exemplo se $n = 1024$, $m = 32$, e $d = 4$ então $s = 240$ e $Y_0 = y_0y_1\cdots y_{303}$, $Y_1 =$

$y_{240}y_{241}\cdots y_{543}$, $Y_2 = y_{480}y_{481}\cdots y_{783}$ e $Y_3 = y_{720}y_{721}\cdots y_{1023}$. Podemos garantir nesse exemplo que qualquer cadeia de caracteres com pelo menos $2m = 64$ caracteres em Y está em pelo menos um dos Y_i.

Podemos calcular o $ASM(X, Y)$ da seguinte maneira:

- **Passo 1:** Cada Bloco(i) lê X e Y_i da memória global e escreva na memória compartilhada.
- **Passo 2:** Cada Bloco(i) calcula o valor de $ASM(X, Y_i)$ em paralelo.
- **Passo 3:** cada Bloco(i) escreve o valor de $ASM(X, Y_i)$ na memória global.
- **Passo 4:** É calculado o $mínimo\{ASM(X, Y_i) \mid 0 \leq i \leq d-1\}$

Com isso consegue-se encontrar o $ASM(X, Y)$ de forma paralela em uma GPU. O trabalho [2] mostra por análise que o modelo utilizado é ótimo em tempo.

3.2.4 Distância de Edição Aproximada Utilizando Comunicação *Intra-Warp*

Este algoritmo possui como característica principal a utilização de comandos *shuffle* a fim de evitar o uso de memória compartilhada para comunicações *inter-warp*.

O algoritmo apresentado na Seção 3.2.3 apresenta algumas limitações. A primeira delas é no Algoritmo 3.6, há um desperdício de recursos no primeiro `if` e no `else if` seguinte, visto que estão dentro de um *loop* e são checados a cada iteração. A melhor forma seria inicializar a matriz com esses valores e deixar o loop apenas para calcular valores da matriz.

Um segundo ponto se deve ao algoritmo utilizar tanto X como Y na memória compartilhada. A placa utilizada foi a GTX 580 [40] que por se tratar de uma arquitetura Fermi tem uma memória compartilhada de 48 KB por Bloco [22].

Se considerarmos apenas o Y, temos 4MB para a memória compartilhada, teríamos que ter pelo menos $\frac{4M}{48KB} = \approx 85$ blocos para armazenar todo o conteúdo de Y.

Além das limitações citadas, o algoritmo poderia ser otimizado se utilizasse mais registradores, por exemplo em vez de colocar X na memória compartilhada cada *thread* i salvaria o caractere x_i em um registrador.

Algoritmo w-SCAN

o w-SCAN visa reduzir o tempo total gasto para se calcular o casamento de padrões com busca aproximada ou ASM. Para isso nosso foco foi reduzir o custo de comunicação, no

O-ASM além da leitura de X e Y e o retorno do resultado final todas comunicações que ocorrem são feitas na matriz e que pertence a memória compartilhada.

Toda *thread* precisa ler 3 posições da memória compartilhada e escrever em uma posição em cada iteração, isso pode ser muito custoso visto que é feito por todas as *threads* e a utilização de registradores ao invés de memória compartilhada poderia evitar problemas de colisão no acesso e até mesmo se mostrar mais rápido naturalmente.

o w-SCAN visa reduzir o uso da memória compartilhada e utilizar registradores para manipular os valores que antes ficavam nessa memória, para isso precisamos de algo que substitua a comunicação com a memória compartilhada.

A implementação chamada de w-SCAN divide o que seria a matriz c do algoritmo sequencial (Seção 3.2.1) em $\frac{m}{w}$ partes, logo cada parte teria w linhas, cada uma dessas partes seria responsável por uma *warp*. O algoritmo não precisa utilizar memória compartilhada para armazenar a matriz, apenas precisamos de memória compartilhada para passar valores entre *threads* de *warps* diferentes.

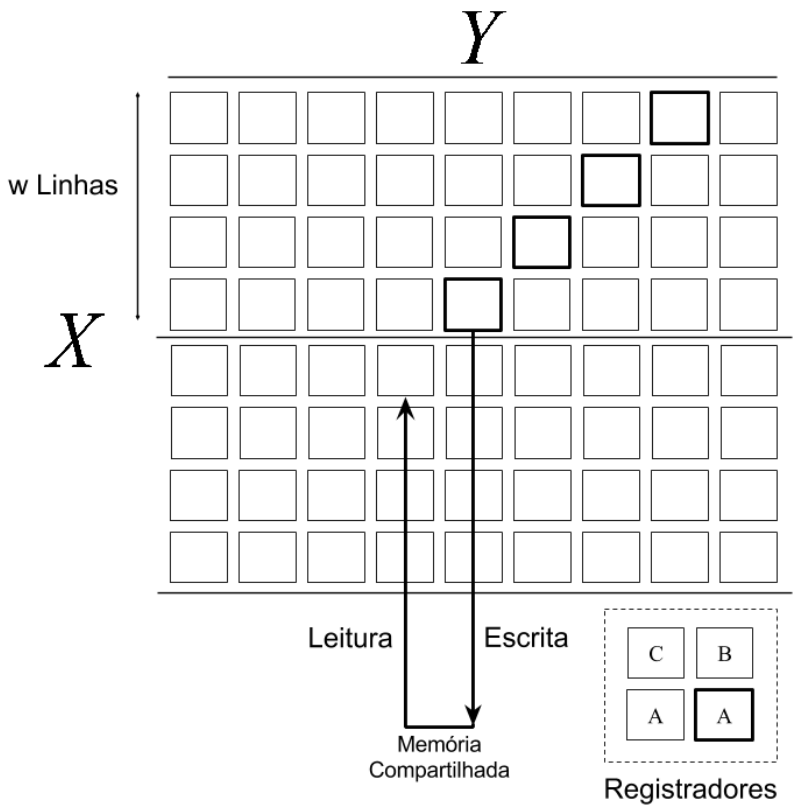


Figura 3.7: Funcionamento do w-SCAN com $w = 4$.

A Figura 3.7 mostra o funcionamento do algoritmo para o tamanho da *warp* $w = 4$, cada *thread* fica responsável por uma linha e somente em uma parte da figura é necessário o uso da memória compartilhada, ou seja, no caso do desenho, só é necessário um espaço

de memória compartilhada para transferir os valores de uma *warp* para outra durante todo o funcionamento do algoritmo. Dessa forma só será necessário $\frac{m}{w}$ variáveis de memória compartilhada.

A comunicação entre *threads* de uma mesma *warp* ocorre de uma forma diferente. A Figura 3.7 mostra que em um determinado momento o A representa o vizinho da esquerda, o B o de cima e o C da diagonal, o valor calculado usando estes 3 registradores será salvo em A na próxima iteração. No código vemos que B de cada *thread* é na realidade o A da *thread* imediatamente anterior, por isso consegue-se utilizar a instrução *shuffle* para que cada *thread* pegue o valor de A da *thread* vizinha e salve em B. Assim todas *threads* com exceção da primeira de cada *warp* recebem a variável A da *thread* anterior e salvam em B, o vizinho C seria o B de uma iteração anterior, o mesmo ocorre com o vizinho A. Então a cada iteração o B é retornado pelo registrador A da *thread* anterior, o C é o B da iteração anterior e o A foi o resultado da iteração anterior.

Implementação do w-SCAN

Algoritmo 3.7: Pseudocódigo do algoritmo w-SCAN.

```
1  i ← ID
2  A ← i
3  B ← i − 1
4  regX ← X[i]
5  for k ←1 até m + n − 1 do
6  |  j ← k − ID
7  |  if 1 ≤ j ≤ n then
8  |  |  C ← B
9  |  |  B ← shfl_up(A)
   |  |  // pega dado da warp anterior
10 |  |  if ID mod w = 0 then
11 |  |  |  B ← f[ID/w]
12 |  |  end
13 |  |  A ← min(A + 1, B + 1, C + (regX ≠ y_j))
   |  |  // salva dado para warp seguinte
14 |  |  if ID mod w = w − 1 then
15 |  |  |  f[ID/w + 1] ← A
16 |  |  end
17 |  end
18 |  syncThreads()
19 end
```

O w-SCAN não precisa de memória compartilhada para obter dados que estejam na mesma *warp*, o Algoritmo 3.7 mostra o funcionamento do w-SCAN. Na linha 10 do código

que somente a primeira *thread* de cada *warp* faz a leitura da memória compartilhada e somente a última *thread* de cada *warp* escreve, como mostrado na linha 14. Para não ter que utilizar uma matriz para salvar os dados, cada *thread* só precisa de 3 registradores: A, B e C.

O algoritmo utiliza menos memória compartilhada por não armazenar uma matriz como o O-ASM, mas podemos ver pelos códigos que nosso algoritmo também reduz substancialmente o número de acessos a essa memória. A Tabela 3.3 mostra a quantidade de acessos requeridos por cada algoritmo para leitura e escrita. Vemos que no caso de escritas o algoritmo O-ASM utiliza w vezes mais operações do que o w-SCAN, esse número sobe para $3w$ quando se trata de operações de leitura.

Tabela 3.3: Número de operações na memória compartilhada.

Algoritmo	**Memória Compartilhada**	
	Operações de Leitura	**Operações de Escrita**
w-SCAN	$(\frac{m}{w}) \times (n + m - 1)$	$(\frac{m}{w}) \times (n + m - 1)$
O-ASM	$3m \times (n + m - 1)$	$m \times (n + m - 1)$

Capítulo 4

Proposta: Casamento de Padrões Aproximado

Essa seção mostra uma proposta para acelerar o casamento aproximado de padrões, o trabalho tem como foco explorar funcionalidades de GPUs recentes, em particular a utilização da comunicação *intra-warp* e a utilização de menos sincronização entre as *threads* como possibilidade de redução do tempo de execução do ASM.

4.1 *w-Shuffle*

O *w-Shuffle* é uma das contribuições da dissertação, este algoritmo se utiliza de menos acessos a memória compartilhada e menos sincronizações que o w-SCAN, ele foi apresentado em [19].

4.1.1 Algoritmo *w-Shuffle*

De forma similar ao w-SCAN [18], este algoritmo divide a matriz d em em $\frac{m}{w}$ faixas com w linhas cada. Cada *thread* processa um símbolo de X e além dos registradores A, B, C e D também se utiliza dos registradores $regWrite$ e $regRead$ para transferir valores do/para o *array* f após w iterações. Ou seja, ao invés de se transferir valores entre as *warps* a cada iteração o w-Shuffle transfere w valores a cada w iterações. Em outras palavras a i-ésima *warp* está a frente da $(i+1)$-ésima *warp* em w iterações. $(0 \leq i \leq \frac{m}{w})$.

A Figura 4.1 mostra a execução do w-Shuffle com $m = 8$ and $w = 4$. Note que os w valores na última linha de cada faixa é movida utilizando operações `shfl_down` para os registradores $regRead$ e $regWrite$. Após w iterações, A (i)-ésima faixa colocaria os valores armazenados nos registradores $regWrite$ para o *array* f. Da mesma forma, a $(i+1)$-ésima faixa lê esses valores e armazena eles no registrador $regRead$. Com isso a

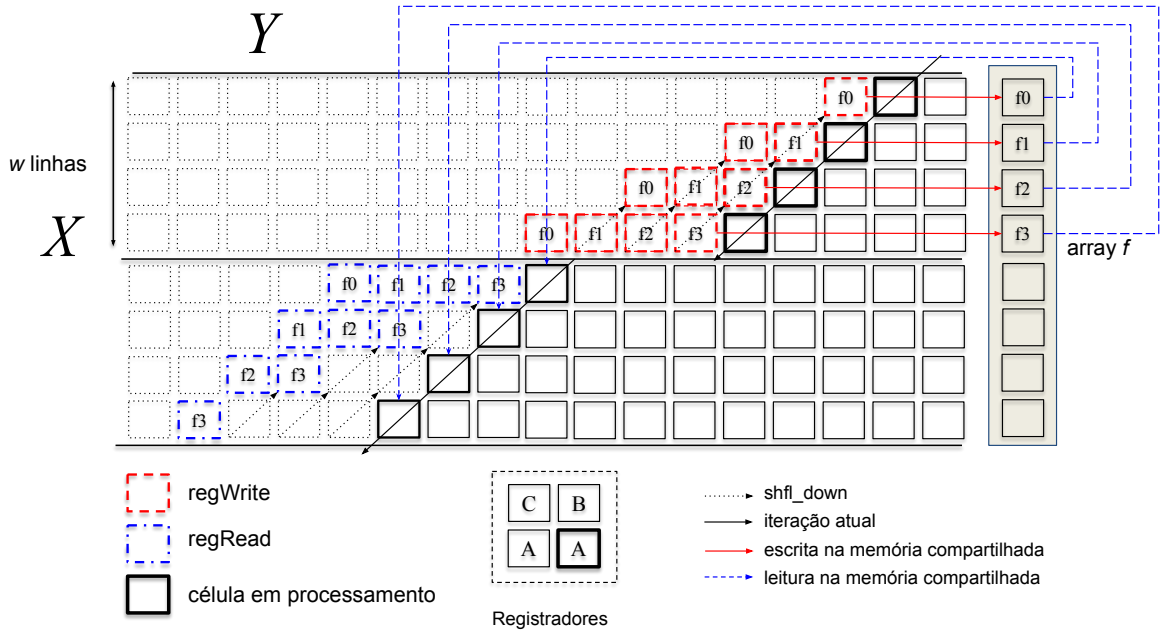


Figura 4.1: Duas partes de w linhas. Quadrados em negrito são os que estão em processamento. O registrador A processa o valor da matriz.

faixa de cima escreve valores no *array* f e a faixa de baixo lê estes valores no mesmo *array*. é importante ressaltar que os valores finais não usaram o *array* f mas sim o registrador A, uma vez que todos os valores de d forem computados, a última *thread* da faixa $(\frac{m}{w} - 1)$ vai calcular os possíveis resultados de ASM(X, Y).

O pseudocódigo do algoritmo pode ser visto na Figura 4.2. Note que a $(i + 1)$-ésima faixa começa após $2w$ iterações da faixa anterior ter começado, $0 \leq i < \frac{m}{w}$). Isso é necessário para permitir que a última *thread* da faixa i calcular w células. Após isso os valores são copiados para o array f e lidos pela $(i + 1)$-ésima faixa. Note que, nessa operação, todas as *threads* da i-ésima *warp* participam da escrita e de forma similar todas as *threads* da $(i + 1)$-ésima *warp* participam da leitura. Ou seja, tanto a leitura quanto a escrita é feita em uma transferência (Seção 2.3.1).

Diferente do w-SCAN o w-Shuffle utiliza dois *loops* for, sendo o *loop* externo (linhas 6 a 29) pra utilização de memória compartilhada (comunicação inter-*warp*) e o *loop* interno (linhas 10 a 27) para processamento e comunicação com comandos shuffle (comunicação intra-warp).

Ao invés de acessar a memória compartilhada a cada iteração é somente feito o acesso tanto para leitura quanto para a escrita a cada w iterações (*loop* externo), o que garante isso é o armazenamento dos valores a serem passados por memória compartilhada nos registradores $regRead$ e $regWrite$.

Podemos ver um exemplo da utilização $regWrite$ na primeira *warp* da Figura 4.1. O

Algoritmo w-Shuffle

```
1  warp_num ← ⌊ID/w⌋
2  pos ← (2w − 1) · warp_num + (ID mod w)
3  A ← i;
4  B ← i − 1;
5  X ← X[i];
6  for k ←1 to m + n − 1 em paralelo
7  begin
8      syncthreads();
9      regRead ← f[ID];
10     for l ←k to k + w − 1 em paralelo
11     begin
12         j ← l − pos;
13         C ← B;
14         B ← shfl_up(A);
15         if (1 ≤ j ≤ n) then
16             //primeira linha da faixa
17             if ((ID mod w) = 0) then
18             B ← regRead;
19             A ← min(A + 1, B + 1, C + (X ≠ y_j));
20         end
21         regWrite ← shfl_down(regWrite);
22         regRead ← shfl_down(regRead);
23         //ultima linha da faixa
24         if ((ID mod w) = w − 1) then
25             regWrite ← A;
26         end
27     end
28     f[ID + w] ← regWrite;
29 end
```

Figura 4.2: Pseudocódigo do w-Shuffle.

valor anteriormente calculado é passado para *thread* imediatamente superior utilizando a instrução `shfl_down()` (linha 22) e o novo valor é sempre atribuído a última *thread* da *warp* (linha 26) , dessa forma como mostrado na figura, no final do *loop* interno teremos os valores processados armazenados no $regWrite$ de cada *thread* de forma ordenada. Ao fim do *loop* interno todas as *threads* escrevem na memória compartilhada ao mesmo tempo (linha 28), o que é feito como se fosse apenas um acesso.

Ao inicio do próximo *loop* os valores na memória compartilhada são lidos e colocados em $regRead$ por todas as *threads* da *warp* também ao mesmo tempo (linha 9) como mostrado na segunda *warp* da Figura 4.1. Apenas quem precisa desses valores é a primeira *thread* da warp, então a cada iteração esta *thread* utiliza o valor de $regRead$ (linha 19) e de

forma similar ao $regWrite$ os valores vão sendo transferidos pela instrução `shfl_down()` (linha 23).

Perceba que o algoritmo acessa o array f sem conflitos, e necessidade de $(\frac{m}{w})$ operação de leitura e a mesma quantidade para escrita em f. O *loop* externo tem $m + n - 1$ iterações, logo o algoritmo necessita de $((\frac{m}{w}) \times (m + n - 1))$ operações de leitura e escrita na memória compartilhada para transferir valores entre as $(\frac{m}{w})$ *warps* enquanto calcula o ASM(X, Y). Perceba que o número de instruções de leituras e escritas executas são incrementadas por *warp* em um MP. Ou seja, as transferências para o array f são feitas em blocos de w valores de uma vez. As w *threads* em uma *warp* realizam a leitura e a escrita em uma instrução apenas. Logo o w-Shuffle utiliza $((\frac{m}{w}) \times (\frac{m+n-1}{w}))$ operações de escrita e $((\frac{m}{w}) \times (\frac{m+n-1}{w}))$- operações de leitura. Comparado ao w-SCAN, o número de acessos a memória compartilhada é reduzido em w vezes. Claro que para reduzir esse número de acessos o algoritmo precisa de um pequeno atraso para transferir os w valores de uma vez só entre as *warps*. Depois veremos que esse pequeno atraso é compensado pela redução no número de sincronizações necessárias.

4.1.2 Resultados

Os experimentos foram realizados em uma GeForce GTX 960 GPU [30]. Esta placa tem 8 SM, tamanho da *warp* w de 32 e 2 GB de memória. Comparamos o w-Shuffle com o w-Scan. Utilizamos o compilador nvcc versão 7.5.17, versão CUDA 7.5 e o Sistema operacional Arch Linux 4.6.3.1.

A entrada Y foi particionada em $D = 128, 256, 512, 1024, 2048$ partes e D blocos, m *threads* foram usadas para calcular ASM$(X, Y_i)(0 \leq i \leq D - 1)$. Assim como o O-ASM da Seção 3.2.3 os caracteres de X e Y foram tratados como *unsigned char* de 8 bits. Os valores de X e Y são randômicos 0 ou 1, logo "$x_i \neq y_j$" tem uma probabilidade de $\frac{1}{2}$ de ser verdadeiro.

O tempo de computação da GPU se refere a execução do *kernel*, excluímos o tempo de transferência de dados entre o *host* e o *device*, que no caso do algoritmo era ínfimo. A GPU processa os dois algoritmos e calcula o ASM$ASM(X, Y_i)(0 \leq i \leq D - 1)$ usando D blocos. O tempo de processamento é contado até antes do ASM(X, Y_i) valore serem transferidos para CPU, onde o resultado é computado.

A Tabela 4.1 mostra o tempo de processamento para o w-Shuffle e o w-SCAN e o respectivo ganho. A tabela mostra o tempo de execução em milissegundos para Y contendo $4M(= 2^{22})$ caracteres e X contendo 2^i caracteres, onde $(5 \leq i \leq 10)$. Perceba que o tamanho de X é relativo ao número de *threads* em cada bloco, que é limitado a 1024 na GPU GTX 960. O tempo de execução mais baixo dos algoritmos é representado em negrito. Como podemos ver, os melhores resultados ocorrem quando $D = 128$. Ow-Shuffle

Tabela 4.1: Tempo de processamento (em ms) para o w-*Shuffle* e o w-SCAN com um número variável de blocos D em uma GTX 960 com o parâmetro $|Y| = 2^{22}$.

$\|X\|$	**Algoritmo**	**Número de Blocos** D					**Ganho**
		$D = 128$	$D = 256$	$D = 512$	$D = 1024$	$D = 2048$	
32	w-Shuffle	3,79	**2,67**	2,68	2,70	2,89	1,84
	w-SCAN	9,45	5,19	4,99	5,20	**4,90**	
64	w-Shuffle	5,61	**5,44**	6,13	5,78	6,08	1,31
	w-SCAN	11,52	**7,13**	8,60	8,17	8,52	
128	w-Shuffle	**10,90**	11,04	12,65	12,07	13,14	1,51
	w-SCAN	17,49	**16,51**	17,29	17,62	18,05	
256	w-Shuffle	**21,99**	22,48	26,09	25,68	29,45	1,45
	w-SCAN	**31,87**	32,69	36,36	35,35	39,92	
512	w-Shuffle	**44,71**	46,61	55,91	58,10	73,73	1,53
	w-SCAN	**68,44**	69,96	77,81	88,39	93,92	
1024	w-Shuffle	**92,43**	100,38	118,00	147,73	210,90	1,73
	w-SCAN	163,07	**159,93**	162,88	221,72	295,67	

provê um ganho acima de 1,3 vezes em relação ao w-SCAN em todos os casos. A coluna "Ganho" mostra a razão dos melhores tempos de execução de ambas implementações.

Com apenas uma faixa, ou seja, $|X| = 32$, o w-Shuffle provê um ganho entre 1.31 e 1.84 comparado ao w-SCAN. w-Shuffle é pelo menos 1.3 vezes mais rápido que o seu competidor, isso ocorre com $|X| = 64$. Perceba que, neste caso, o número de faixas é $\frac{m}{w} = 2$. Quando o número de faixas aumenta, a razão do desempenho entre os algoritmos aumenta a favor do w-Shuffle. Com $|X| = 1024$, $\frac{m}{w} = 32$ faixas, este algoritmo provê uma performance 1,7 vezes maior que seu concorrente.

A Tabela 4.2 mostra o número de instruções de escrita para o w-Shuffle e o w-SCAN, a Tabela 4.3 mostra o número de instruções de escrita. Estes resultados foram adquiridos com a ferramenta de *profiler* "nvprof" [31] na versão 7.5.18 como mostrado na Seção 2.3.4. É importante ressaltar que o número de instruções executadas, mostradas nas tabelas, é incrementado por warp. Por exemplo, se w *threads* em uma mesma *warp* realizam acesso a memória ao mesmo tempo, isso é contato como uma instrução. Ou seja, o número de instruções é contado por *warp* e não por acessos de threads. Como podemos ver na tabela, o w-SCAN precisa de ≈ 32 vezes mais instruções para calcular o ASM quando comparado ao w-Shuffle. Esses resultados confirmam que o w-Shuffle reduz o número de leituras e escritas na memória compartilhada em uma razão de w.

O w-Shuffle e o w-SCAN não ultrapassam o limite de 32 registradores por thread, eles usam 26 e 21 registradores, respectivamente. No caso de 32 *threads* por bloco, os dois algoritmos possuem uma ocupação de 50%, devido a limitação dos blocos por MP, como mostrado na Seção 2.3.3. A partir de 64 threads, a ocupação dos dois algoritmos atinge

Tabela 4.2: Número de instruções de escrita executadas (em milhões de instruções).

	D=128		D=256		D=512		D=1024		D=2048		Razão	
$\|X\|$	w-Shuffle	w-SCAN	w-Shuffle	w-SCAN	w-Shuffle	w-SCAN	w-Shuffle	w-SCAN	w-Shuffle	w-SCAN	Min	Max
32	0,13	4,21	0,13	4,23	0,13	4,26	0,14	4,33	0,14	4,46	30,93	32,77
64	0,26	8,44	0,27	8,50	0,27	8,60	0,28	8,81	0,29	9,24	31,46	31,86
128	0,53	16,97	0,54	17,15	0,55	17,53	0,58	18,28	0,64	19,79	30,92	31,87
256	1,08	34,26	1,10	34,96	1,16	36,37	1,26	39,19	1,47	44,83	30,50	31,78
512	2,20	69,81	2,30	72,54	2,51	77,98	2,92	88,87	3,74	110,62	29,58	31,54
1024	4,60	144,84	5,00	155,53	5,81	176,90	7,44	219,64	10,68	305,14	28,57	31,11

Tabela 4.3: Número de instruções de leitura executadas (em milhões de instruções),

	D=128		D=256		D=512		D=1024		D=2048		Razão	
$\|X\|$	w-Shuffle	w-SCAN	w-Shuffle	w-SCAN	w-Shuffle	w-SCAN	w-Shuffle	w-SCAN	w-Shuffle	w-SCAN	Min	Max
32	0,13	4,21	0,13	4,23	0,13	4,26	0,14	4,33	0,14	4,46	30,93	32,77
64	0,26	8,44	0,27	8,50	0,27	8,60	0,28	8,81	0,29	9,24	31,46	31,86
128	0,53	16,97	0,54	17,15	0,55	17,53	0,58	18,28	0,64	19,79	30,92	31,87
256	1,08	34,26	1,10	34,96	1,16	36,37	1,26	39,19	1,47	44,83	30,50	31,78
512	2,20	69,81	2,30	72,54	2,51	77,98	2,92	88,87	3,74	110,62	29,58	31,54
1024	4,60	144,84	5,00	155,53	5,81	176,90	7,44	219,64	10,68	305,14	28,57	31,11

100%. O uso da memória compartilhada por bloco dos dois algoritmos é muito inferior ao limite por bloco.

Esse capítulo mostrou uma técnica de casamento de padrões aproximado com a utilização de comando *shuffle* com o intuito de reduzir o número de acessos a memória compartilhada e as sincronizações entre threads. Muitas vezes necessitamos fazer uma busca por casamentos exatos, o capítulo seguinte vai apresentar uma proposta nesse tema.

Capítulo 5

Proposta: Casamento de Padrões Exato

Essa seção mostra uma proposta que utiliza a soma de prefixos para o algoritmo de Rabin-Karp que foi apresentado na Seção 3.1.2, na Seção 5.1 é apresentado uma adaptação do Rabin-Karp original para utilização de paralelismo, são mostrados alguns resultados e uma discussão de sua eficiência. Na Seção 5.2 é apresentado uma alternativa utilizando soma de prefixos, ao final da seção os resultados são comparados com o Rabin-Karp original.

5.1 Adaptando o Rabin-Karp para Processamento Paralelo

No algoritmo de Rabin-Karp (Seção 3.1.2), o processamento é realizado de forma sequencial e é feita a comparação de apenas um padrão com um texto, neste trabalho utilizamos o Rabin-Karp tradicional, modificando=o para comparamos vários padrões com um texto de forma paralela.

Resolvemos utilizar além da GPU, uma interface de programação para utilização de multi-processos em CPU, essa interface é a *Open Multi-Processing* (OpenMP) [41], com ela conseguimos utilizar múltiplos processos para serem executados de forma paralela em vários núcleos de CPU. Para o caso do Rabin-Karp a implementação em OpenMP possui o mesmo algoritmo da GPU, apenas foi feito outro código devido a mudança de arquitetura.

O Algoritmo 5.1 apresenta a daptação do RK, para adaptar o mesmo para GPU separamos o cálculo do *hash* do padrão em uma função distinta, chamada aqui de `calculateHashPattern`, o calculo do *hash* do texto e a comparação de caracteres é realizado na função `FindMatches`. Neste Algoritmo a variável ID é o identificador da *thread* no contexto, onde cada *thread* tem um ID único.

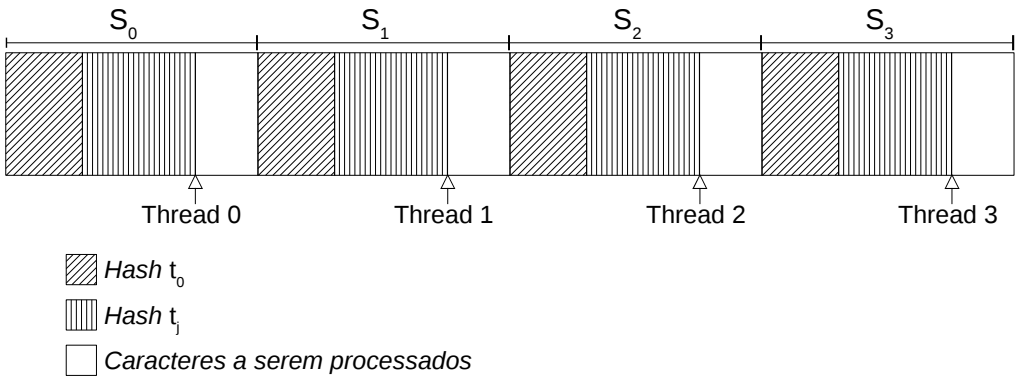


Figura 5.1: Função `FindMatches` executado no RK paralelo.

O cálculo do *hash* dos padrões realizada na função `calculateHashPattern` é feita com uma *thread* para cada padrão, gravando o resultado de cada *hash* na variável HP. Com o *hash* dos padrões calculados executamos a função `FindMatches`, que também é representado na Figura 5.1, para isso dividimos o texto T em S partes, onde $gridDim$ é a dimensão do *grid* executado na GPU, ou seja, é a quantidade total de *threads* executadas naquela função, cada *thread* vai ser responsável pelo processamento de uma parte. No início de cada parte é realizado a inicialização de variáveis nas linhas 7-9 e o cálculo do *hash* inicial t_0 representado no laço da linha 10 do algoritmo. A partir da linha 13 o *hash* t_i é recalculado enquanto a *thread* percorre sua parte no texto S_i, na linha 14 é feita a troca do padrão para verificação de todos os *hashes* de padrões já calculados, no caso de uma ocorrência é feita uma nova verificação, representado no laço da linha 16, onde se verifica caractere a caractere, caso todos os caracteres do texto e do padrão coincidam é relatado o casamento.

5.1.1 Resultados

Para a avaliação do Rabin-Karp resultados a GPU utilizada foi a GeForce Tesla $V100$ [42]. Essa placa possui 84 SM e 16GB de memória. Nos experimentos foi utilizado o compilador nvcc versão 7.5.17, versão do CUDA 9.1. Para comparação os algoritmos também foram implementados de forma sequencial e paralela na CPU utilizando o OpenMP e linguagem $C++$ em um Intel(R) Xeon(R) CPU E7-8870 v4 2.10GHz usando o compilador g++ versão 4.8.5. Os códigos rodando OpenMP usaram 128 *threads* e foi escolhido os escalonamentos que apresentaram o emlhor tempo, a GPU utilizou 128 blocos com 256 *threads* cada na função `FindMatches`. Os resultados são uma média de 20 de execuções, e foi desconsiderado o tempo de transferência de dados entre a CPU e a GPU.

Algoritmo 5.1: Algoritmo de Rabin-Karp paralelo.

```
   Entrada: P,T,D,Q
   // P string do padrão
   // T string do texto
   // D tamanho do alfabeto (|Σ|)
   // Q número primo
   // N ← |T|
   // M ← |P|
   // K quantidade de padrões
   // H ← d^{M-1} mod q
 1 Function calculateHashPattern (P[K][M],HP[K])
 2 |   for i ← 0 até M − 1 do
 3 |   |   p ← (dp + P[ID][i]) mod q
 4 |   end
 5 |   HP[ID]=p
 6 Function FindMatches (T,HP[K],P)
 7 |   t ← 0
 8 |   tamanhoParte ← N / gridDim
 9 |   inicio ← ID * tamanhoParte
10 |   for i ← 1 até m do
11 |   |   t ← (dt + T[i]) mod q
12 |   end
13 |   for i ← inicio até start + tamanhoParte do
14 |   |   for k ← 0 até K − 1 do
15 |   |   |   if t ← (HP[k]) then
16 |   |   |   |   if P[k][1] .. P[k]][M]=T[i] .. T[i+M-1] then
17 |   |   |   |   |   Casamento na posição i com padrão k
18 |   |   |   |   end
19 |   |   |   end
20 |   |   end
21 |   |   t ← (D(t − T[i + 1]h) + T[i + M + 1]) mod q
22 |   end
23 HP=calculateHashPattern(P)
24 FindMatches(T,HP,P)
```

Nos experimentos, nos consideramos $k = 1, 4, 16, 64, 256$ padrões com $m = 10, 15, 20, 25, 30$ caracteres cada, o texto n tem 2^{27} caracteres (≈ 128 *Mbytes*). m e n são definidos aleatoriamente como 0 ou 1.

A Tabela 5.1, mostra os resultados em milissegundos para a versão sequencial, a versão em OpenMP e a versão em GPU do Rabin Karp com múltiplos padrões. Note que quando o valor de k é aumentado o tempo de execução também aumenta, e a variação de k influencia o tempo de execução muito mais que a variação de m

A Tabela 5.2 mostra os ganhos entre as implementações apresentadas, conseguimos ver uma melhora significativa quando paralelizamos a versão sequencial para $OpenMP$ onde temos um ganho que vária próximo a 50 em todos os casos, o aumento parecido para todos os casos é em decorrência dos dois casos utilizarem a CPU e consequentemente fora a quantidade de processadores, temos a utilização de memória e processamento parecidos.

Tabela 5.1: Resultados do Rabin-Karp nas 3 implementações (ms)

		$m = 10$	$m = 15$	$m = 20$	$m = 25$	$m = 30$
Sequencial	$k = 1$	716,20	712,60	708,69	714,95	712,62
	$k = 4$	711,62	700,39	719,10	719,84	719,52
	$k = 16$	1887,04	1812,51	1825,72	1826,54	1860,93
	$k = 64$	6391,86	6282,33	4247,17	4227,28	6502,15
	$k = 256$	17361,80	17195,05	16906,69	16884,93	19508,51
OpenMP	$k = 1$	13,90	13,12	12,95	14,39	12,79
	$k = 4$	22,48	20,03	16,95	14,26	18,38
	$k = 16$	35,57	39,51	29,30	28,37	29,84
	$k = 64$	105,59	103,59	102,05	100,28	101,58
	$k = 256$	419,69	420,01	361,19	367,78	359,35
GPU	$k = 1$	9,13	9,50	10,05	11,49	10,83
	$k = 4$	10,37	10,82	11,48	11,97	12,32
	$k = 16$	11,37	11,66	12,32	12,76	13,44
	$k = 64$	18,05	15,22	15,28	15,55	15,97
	$k = 256$	52,56	47,54	44,54	45,99	46,42

Quando comparamos a versão da GPU com a OpenMP temos ganhos entre 1,18 e 1,52 para $k = 1$ e esse ganho fica maior cada vez que aumentamos o valor de k até conseguirmos um ganho entre 7,74 e 8,84 para $k = 256$. Diferente do que ocorre com a versão em OpenMP comparada a sequencial, quando aumentamos k, o ganho da GPU em relação as outras implementações cresce. Para um k maior temos um número maior de comparações, consequentemente temos um maior número de acessos a memória, esse aumento de ganho com o número maior de comparações indica um ganho no acesso a memória no caso da GPU.

5.1.2 Discussão

O algoritmo de Rabin-Karp para múltiplos padrões apresenta uma perda de desempenho quando aumentamos o k, o motivo fica claro no laço da linha 14 do Algoritmo 5.2, nesse algoritmo é feita uma nova comparação dos *hashes* para os k padrões existentes, é possível melhorar esse número de comparações utilizando uma tabela *hash*, que é uma estrutura de dados que associa chaves de pesquisa a valores.

Ao ver a Figura 5.1 notamos que cada *thread* deve calcular o *hash* t_0 dos primeiros m caracteres, esse processo está representado na linha 10 do Algoritmo 5.2. A partir de t_0 calculado o *hash* é recalculado desconsiderando o último caractere e adicionando o novo, mostrado na linha 21 do algoritmo, ou seja, para cada *thread* temos um *overhead* de m caracteres. O *hash* da posição atual da *thread* depende do *hash* da posição anterior. Caso os *hashes* de cada posição fossem independentes, conseguiríamos uma paralelização

Tabela 5.2: Ganho das diferentes implementações no Rabin-Karp

		$m = 10$	$m = 15$	$m = 20$	$m = 25$	$m = 30$
OpenMP vs Sequencial	$k = 1$	51,51	54,32	54,73	49,67	55,73
	$k = 4$	31,65	34,97	42,42	50,47	39,15
	$k = 16$	53,05	45,88	62,30	64,39	62,37
	$k = 64$	60,53	60,65	41,62	42,15	64,01
	$k = 256$	41,37	40,94	46,81	45,91	54,29
GPU vs Sequencial	$k = 1$	78,47	75,02	70,55	62,21	65,82
	$k = 4$	68,66	64,72	62,65	60,15	58,41
	$k = 16$	165,97	155,38	148,22	143,18	138,48
	$k = 64$	354,07	412,85	277,93	271,81	407,11
	$k = 256$	330,29	361,73	379,58	367,16	420,28
GPU vs OpenMP	$k = 1$	1,52	1,38	1,29	1,25	1,18
	$k = 4$	2,17	1,85	1,48	1,19	1,49
	$k = 16$	3,13	3,39	2,38	2,22	2,22
	$k = 64$	5,85	6,81	6,68	6,45	6,36
	$k = 256$	7,98	8,84	8,11	8,00	7,74

completa e sem *overheads* para os cálculos, podemos tornar esse cálculo independente utilizando soma de prefixos, que será apresentado na seção seguinte.

5.2 Utilização de Soma de Prefixos para o Rabin-Karp

Nossa proposta visa diminuir o tempo de processamento do cálculo do Rabin-Karp. Para tal, utilizaremos o teorema abaixo como base para nossa proposta.

Teorema 5.1. *Para quaisquer dois números primos d e q, $d^{q-1} \bmod q = 1$ é sempre verdadeiro.*

Por exemplo, para $d = 2$ e $q = 13$, $d^{q-1} \bmod q = 2^{12} \bmod 13 = 1$. Para este teorema, $d^i \bmod q = d^{i+(q-1)} \bmod q$ é verdadeiro. Com isso temos o seguinte corolário:

Corolário 5.2. *Para quaisquer dois números primos d e q, e um inteiro i, $d^i \bmod q = d^{i \bmod (q-1)} \bmod q$ é sempre verdadeiro.*

Por exemplo, para $d = 2$, $i = 15$, e $q = 13$, $d^{15} \bmod q = 8$ e $d^{15 \bmod (13-1)} = 2^3 \bmod 13 = 8$. Para $T = t_0 t_1 \dots t_{n-1}$ dizemos que $a_i = d^{n-i-1} t_i$ para todo $i (0 \leq i \leq n-1)$ e $\hat{a}_i = a_0 + a_1 + \dots + a_i$ é a soma de prefixos de a. Ou seja, $\hat{a}_i = d^{n-1} t_0 + d^{n-2} t_1 + \dots + d^{n-i-1} t_i$. Se tivermos todos as somas de prefixos $\hat{a}_0, \hat{a}_1, \dots, \hat{a}_{n-1}$, podemos calcular o valor de *hash* $h(t_j t_{j+1} \dots t_{j+m-1})$ pela seguinte equação:

$$h(t_j t_{j+1} \dots t_{j+m-1}) = (\hat{a}_{j+m-1} - \hat{a}_{j-1}) \cdot d^{m-n+j} \tag{5.1}$$

Visto que

$$\begin{aligned}
\hat{a}_{j+m-1} - \hat{a}_{j-1} &= a_j + a_{j+1} + \dots + a_{j+m-1} && (5.2)\\
&= d^{n-j-1}t_j + dn - j - 2t_{j+1} + \\
&\quad \dots + d^{n-j-m}t_{j+m-1} && (5.3)\\
&= (d^{m-1}t_j + d^{m-2}t_{j+1} + \\
&\quad \dots + d^0 t_{j+m-1}) \cdot d_{n-j-m} && (5.4)\\
&= h(t_j t_{j+1} \dots t_{j+m-1}) \cdot d^{n-m-j}, && (5.5)
\end{aligned}$$

podemos confirmar que a Equação 5.1 está correta, perceba que $m - n + j$ pode não ser positivo.

lookup table

Tabela 5.3: Módulos para $d = 2$ e $q = 13$.

i	0	1	2	3	4	5	6	7	8	9	10	11	12
$d^i \bmod q$	1	2	4	8	3	6	12	11	9	5	10	7	1

i	12	13	14	15	16	17	18	19	20	21	22	23	24
$d^i \bmod q$	1	2	4	8	3	6	12	11	9	5	10	7	1

Suponha que o valor de $d^0 \bmod q, d^1 \bmod q, \dots, d^{q-2} \bmod q$ está salvo em um vetor de tamanho $q - 1$, como mostrado na Tabela 5.3. Com este vetor podemos calcular d^i para qualquer inteiro i utilizando o Corolário 5.2. Desde que $0 \leq i \bmod (q-1) \leq q-2$,podemos calcular $d^i \bmod q$ lendo o $(i \bmod (q-1))$-ésimo elemento do vetor. Por exemplo, se $d = 2$, $q = 13$ e $i = 100$, em vez de calcular $d^i \bmod q = 2^{100} \bmod 13$, podemos calcular $i \bmod (q-1) = 100 \bmod 12 = 4$ e acessar a posição 4 do vetor da Tabela 5.3para pegar o resultado final 3. O resultado de $d^i \bmod q$ pode ser obtido da posição $i \bmod (q-1)$ do vetor. Perceba que os valores de $d^i \bmod q$ na Tabela 5.3 sempre repetem para $i > q - 1$, ou seja, só é necessário armazenar q valores.

Passos do Algoritmo

O algoritmo pode ser descrito por meio de 5 passos, como segue:

- **Passo 1** Carregue uma *lookup table* pré-processada para $d^i \bmod q$ $(0 \leq i \leq q-1)$.

- **Passo 2** Calcule os valores de *hash* dos padrões $h(P_k)$ para k $(0 \leq k \leq p-1)$ em paralelo e crie uma *hash table HT* usando os valores calculados.
- **Passo 3** Calcule $a_0, a_1, \ldots, a_{n-1}$ em paralelo.
- **Passo 4** Calcule a soma de prefixos $\hat{a}_0, \hat{a}_1, \ldots, \hat{a}_{n-1}$.
- **Passo 5** Para cada j $(0 \leq j \leq n-m)$, calcule $(\hat{a}_{j+m-1} - \hat{a}_{j-1}) \cdot d^{m-n-j}$, que é igual a $h(t_j t_{j+1} \ldots t_{j+m-1})$. Se $HT(h(t_j t_{j+1} \ldots t_{j+m-1})) \neq -1$ verifique caractere a caractere do texto com o padrão.

Os passos acima podem ser representados no Algoritmo 5.2, a chamada pra cada passo está nas linhas 31 a 35, no caso do primeiro passo a *lookup table*, representada pela variável `LT` é pré-processada e apenas carregada.

Populando a *Hash Table*

No passo 2, representado pela Função `calculateHashTable` do Algoritmo 5.2, nós calculamos a HT considerando que dois padrões podem ter um mesmo valor de *hash*, para isso utilizamos um vetor de controle para verificar se já existe um *hash* com aquela posição. Neste passo os *hashes* de cada padrão p são calculados de forma paralela, para evitar condição de corrida são utilizados dois vetores, um de controle e um sendo a própria HT. A Figura 5.2 mostra um exemplo de inserção de 5 padrões, no primeiro vetor é realizado um `atomicAdd` na posição respectiva ao *hash* daquele padrão, essa instrução permite que uma *thread* incremente o valor de um variável do vetor e receba o valor antigo dela de forma atômica. O vetor de controle é inicializado com zeros e o da *Hash Table* é inicializada com -1.

A Figura 5.2a mostra os vetores inicializados e a inclusão de 3 *hashes*. A instrução `atomicAdd` é utilizada nas posições respectivas aos *hashes* calculados, como o retorno da operação atômica nos 3 casos é 0 as threads gravam o id do padrão na mesma posição do hashTable, como é mostrado em negrito na Figura 5.2b. Nessa mesma figura são inseridos outros dois *hashes* porém o retorno da operação atômica é diferente de 0, com isso as threads repetem a operação na posição seguinte, como é mostrado na Figura 5.2c, no caso de de $h(p_3)$ a próxima posição já retorna 0 e o id 3 é gravado nesta posição. Para o $h(p_4)$ temos um retorno diferente de 0 no *hash* calculado e também na posição seguinte, com isso temos uma gravação duas posições adiante do *hash* calculado.

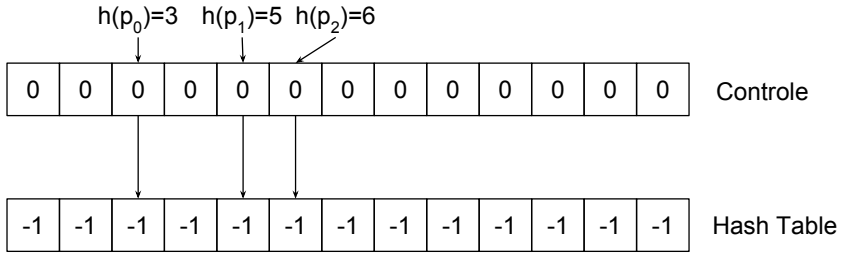
Cálculo da Soma de Prefixos

No passo 3, representado pela função `calculateA` do Algoritmo 5.2, é calculado cada $a_i = d^{n-i-1} \cdot t_i$ de forma paralela. No passo 4 é feito a soma de prefixos dos valores

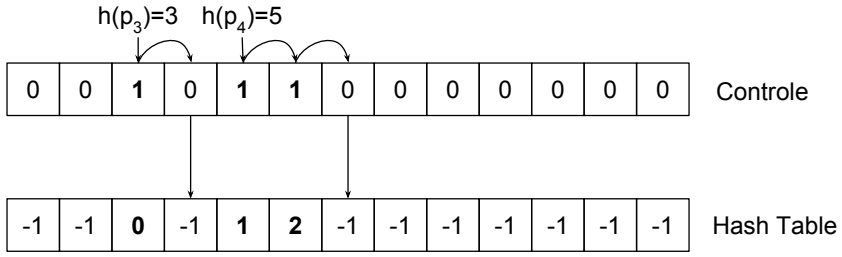
Algoritmo 5.2: Proposta: RK utilizando soma de prefixos.

```
   Entrada: P,T,D,Q
   // P string do padrão
   // T string do texto
   // D tamanho do alfabeto (|Σ|)
   // Q número primo
   // N ← |T|
   // M ← |P|
   // K quantidade de padrões
 1 Function calculateHashTable (P[K][M],HT[Q],control[Q]
 2 |   p ← 0
 3 |   oldValue ← 1
 4 |   for i ← 1 até M incrementando blockDim do
 5 |   |   p ← (D * p + P[ID][i])mod q
 6 |   end
 7 |   while old ≠ 0 do
 8 |   |   oldValue = atomicAdd(control[p], 1)
 9 |   |   if oldValue ≠ 0 then
10 |   |   |   p ← (p + 1) mod Q
11 |   |   end
12 |   end
13 |   HT[p] ← ID
14 Function CalculateA (T[N],LT[Q])
15 |   for i ← 1 até N incrementando gridDim do
16 |   |   A[i] ← LT[(N − j − 1) mod (Q − 1))] * T[i]
17 |   end
18 Function FindMatches (A[N],HT[Q],LT[Q],T[N],P[K][M],control[Q])
19 |   for i ← 1 até N − M + 1 incrementando gridDim do
20 |   |   H ← |Â[M + i] − Â[i − 1]| * LT[(M − N + i) mod (Q − 1)] mod Q
21 |   |   if control[H] ≠ 0 then
22 |   |   |   posHash ← H
23 |   |   |   while controle[posHash] > 0 do
24 |   |   |   |   patternID ← HT[posHash]
25 |   |   |   |   if P[patternID][1]...P[patternID]][M] = T[i]...T[i + M − 1] then
26 |   |   |   |   |   Casamento na posição i com padrão k
27 |   |   |   |   end
28 |   |   |   |   posHash ← (posHash + 1) mod Q
29 |   |   |   end
30 |   |   end
31 |   end
32 LT=Modules()
33 HT=calculateHashTable(P,HT,control)
34 A=calculateA(T,LT)
35 A=calculatePrefixSum(A)
36 FindMatches(A,HT,LT,P,T,control)
```

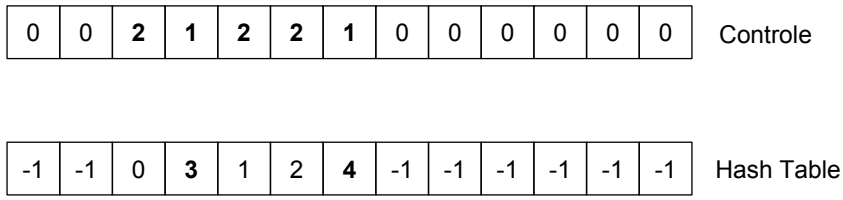
calculados no passo 3, ele é feito de formas paralelas distintas na CPU e na GPU. O caso do OpenMP é exemplificado na Figura 5.3, em um primeiro momento são criados vetores auxiliares como mostrado na Figura 5.3a, considerando um vetor de entrada de tamanho N, o primeiro vetor auxiliar terá tamanho $\frac{N}{2}$, o segundo $\frac{N}{4}$ e último terá tamanho 1, cada célula destes vetores representara a soma de duas células da vetor anterior, ou seja, uma



(a) Inserção dos *hashes* de p_0, p_1 e p_2.



(b) Inserção dos *hashes* de p_3 e p_4.



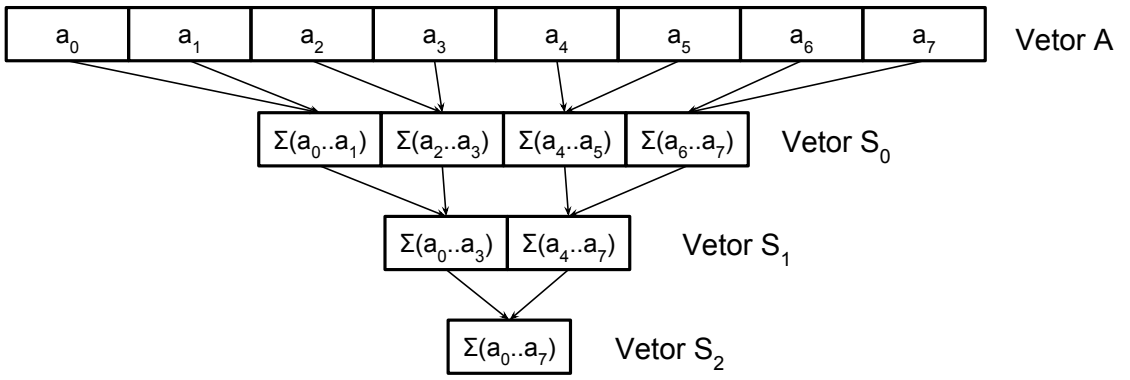
(c) *Hash Table* completa.

Figura 5.2: preenchendo a *Hash Table*

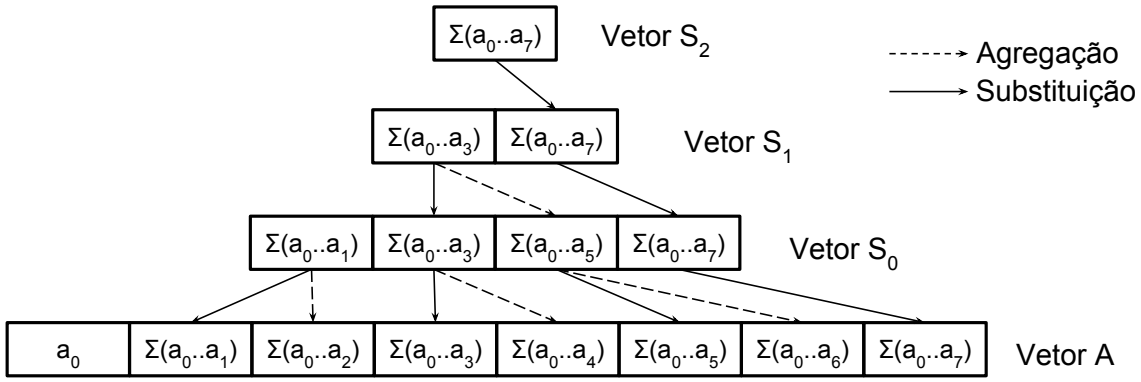
posição i será respectiva a soma das posições $i \cdot 2$ e $i \cdot 2 + 1$ do vetor anterior.

Com os vetores auxiliares calculados fazemos o processo da Figura 5.3b, iniciando do menor vetor para o maior, substituímos o valor da posição $i * 2 + 1$ com a posição i do vetor com id imediatamento superior, e somamos o valor da posição $i * 2 + 2$ com o valor i deste mesmo vetor, dessa forma, quando chegarmos ao último vetor, que é o prório vetor de entrada ele vai ter a soma de prefixos calculado para todo array de tamanho N.

No caso da GPU é utilizada a soma de prefixos da biblioteca CUDA UnBounded (CUB) [43]. CUB é uma biblioteca em $C++$ que provê *kernels* eficientes que podem ser utilizados para diferentes aplicações e arquiteturas de GPU. Neste trabalho utilizamos o algoritmo *"decoupled look-back"* para calcular a soma de prefixos global [44]. Entretando, o código foi modificado para que a soma de dois termos a e b na soma de prefixos fosse calculada usando $(a + b) \bmod q$. Para o algoritmo citado o vetor de entrada é divido em várias partes. No caso do da Figura 5.4 o vetor é dividido em 4 partes, cada uma destas partes vai ser processado por um bloco de *threads*. Cada um destes blocos vai ter duas



(a) Criação dos vetores auxiliares S.



(b) Utilizando os vetores auxiliares S para calcular a soma de prefixos no OpenMP.

Figura 5.3: Soma de prefixos utilizada no OpenMP.

saídas e uma *flag*, as saídas são o `aggregate` que é o valor agregado daquela parte do vetor, e o `incl-prefix` que é o agregado total até aquele bloco. A *flag* indica se esses valores já estão prontos, uma *flag* no estado X indica que nenhum valor está pronto, uma *flag* no estado A indica que o valor agregado daquele bloco está pronto e por fim, uma *flag* com estado P indica que todos valores estão prontos.

Tabela 5.4: Exemplo de execução no CUB [44].

ID da partição	0	1	2	3	4	5	6	7
Flag	P	A	A	P	A	P	X	A
Agregado	2	2	2	2	2	2	-	2
soma de prefixo	2	4	-	8	-	12	-	-

O exemplo da Tabela 5.4 do artigo que o CUB é apresentado [43] mostra uma fotografia no meio da execução do cálculo e uma soma de prefixos, neste exemplo todas as partes tem valor agregado igual a 2, a *flag* P na partição 0 indica que esta já terminou sua execução, as partes 1, 2, 4 e 7 com *flag* A já calcularam seu valor agregado e ainda estão

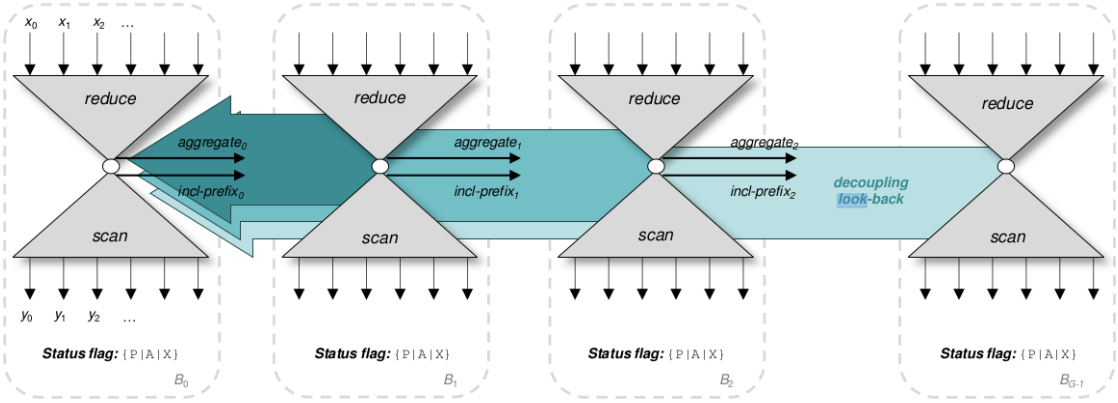


Figura 5.4: Soma de prefixos do CUB [44].

em execução, a parte 3 terminou sua execução utilizando os valores agregados já calculados pelas partes 0, 1 e 2, não precisando esperar o término de todas as partes anteriores, no caso da parte 5 ele utiliza o valor de `incl-prefix` da parte 3 e o valor agregado da parte 4 para finalizar sua execução. Na parte 6 ainda não se calculou o agregado impedindo que a parte 7 termine sua execução. Temos um cálculo completo assim que todas as partes chegam na *flag* P.

Verificação de Casamentos

No passo 5 cada *thread* é independente e é calculado o *hash* de cada parte do texto $h(t_j t_{j+1} \dots t_{j+m-1})$ paralelamente, esse *hash* é calculado através da equação 5.1 utilizando a soma de prefixos calculada no passo 4. Com o *hash* calculado verifica-se se há alguma ocorrência no vetor de controle nesta posição, ou seja, se aquela posição é diferente de 0, caso seja 1 é verificado o casamento somente naquela posição. Caso o valor na posição seja maior que 1 avançamos uma posição no vetor e repetimos o processo de verificação.

Para a checagem do casamento devemos fazer a comparação caractere a caractere e confirmar se há um casamento.

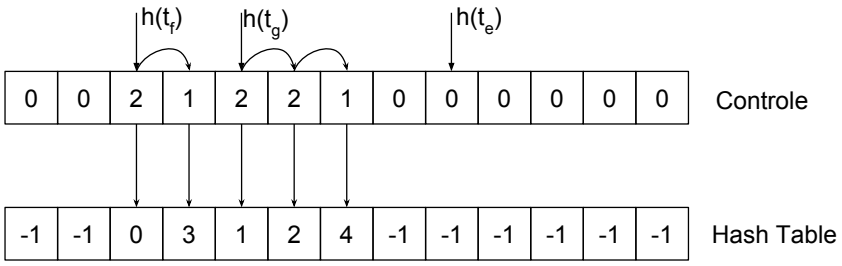


Figura 5.5: Exemplo de verificação de ocorrência nas posições t_e, t_f e t_g.

Temos um exemplo na Figura 5.5, no caso do *hash* da posição t_e do texto temos um valor igual a 0, o que indica que não existe ocorrência de padrão com esse hash, para a posição f devemos verificar a posição calculada e como o retorno é maior que 1 também verificamos a seguinte, e com a mesma lógica verificamos 3 posições com a posição g.

Cada *hash* de padrão calculado será salvo como índice na *hash table* e o ID do padrão será salvo como valor, dessa forma ao procurar um *hash* do texto calculado, conseguimos o verificar se existe padrão com este *hash* e qual seria o ID com apenas um acesso.

Na linha 18 do Algoritmo 5.1 *thread* calcula um termo de $a_i = d^{n-i-1}t_i (0 \leq i < N)$ de forma paralela. Note que a lookup table é utilizada. Com este vetor calculado utilizamos a função de soma de prefixos modificada [43] para obter os $\hat{a}_i (0 \leq i < N)$.

Com o prefix-sum calculado procuramos as ocorrências na linha 29, para isso cada *thread* compara o *hash* relativo a uma posição do texto por vez de forma independente e paralela. O último passo efetuado na linha 32 é calculado da mesma forma que o Algoritmo 5.1 resultando nos casamentos do algoritmo.

$$\begin{aligned}
\hat{a}_{j+m-1} - \hat{a}_{j-1} &= a_j + a_{j+1} + \cdots + a_{j+m-1} && (5.6)\\
&= d^{n-j-1}t_j + dn - j - 2t_{j+1} + \\
&\quad \cdots + d^{n-j-m}t_{j+m-1} && (5.7)\\
&= (d^{m-1}t_j + d^{m-2}t_{j+1} + \\
&\quad \cdots + d^0 t_{j+m-1}) \cdot d_{n-j-m} && (5.8)\\
&= h(t_j t_{j+1} \ldots t_{j+m-1}) \cdot d^{n-m-j}, && (5.9)
\end{aligned}$$

5.3 Resultados

Para a avaliação da nossa proposta, utilizamos o mesmo ambiente da Seção 5.1.1, para a coleta de resultados também utilizamos uma média de 20 execuções. Os tamanho de m e n também são os mesmos.

Os parâmetros de entradas são armazenados na memória global juntamente com a *lookup table* do Passo 1. Os parâmetros de tamanho do alfabeto $d = 2$ e número primo $q = 65521$, que é o maior número primo menor que 2^{16}. No passo 2 utilizamos uma *thread* para cada calculo de *hash* de padrão $h(P_k)$. Nos Passos 3 e 5, para melhorar a ocupação, utilizamos 128 blocos com 256 *threads* cada. No passo 4 utilizamos a soma de prefixos modificada da biblioteca CUDA *UnBounded* (CUB) versão 1.7.3.

Para os resultados a seguir, devemos levar em consideração a quantidade de casamentos e o número de padrões. Probabilisticamente temos $\frac{n}{d^q}$ ocorrências de *hashes* iguais

durante o processamento. *Hashes iguais* não significa que temos um casamento, apenas que será feita a checagem caractere a caractere até se confirmar ou descartar um casamento, conforme foi mostrado nos Algoritmos 5.1 e 5.2. O número de casamentos deve ficar por volta de $\frac{k \cdot n}{d^m}$, ou seja, para $n = 2^{27}$, $d = 2$ e $m = 10$ teremos probabilisticamente cerca de $k \cdot 2^{17}$ casamentos. Para $m = 30$ temos uma chance de $k \cdot 2^{-3}$ de ocorrer um casamento durante toda comparação, para $k = 1$ essa chance seria de $2^{-3} = 12{,}5\%$.

A Tabela 5.5 mostra que em nossa proposta os valores de k influenciam de forma discreta o tempo de processamento, isso devido ao a *Hash Table* implementada que permite a verificação dos *hashes* de forma rápida. Com $k = 256$ e $m = 10$ é onde temos o maior tempo de processamento, independente da implementação, isso ocorre devido a este ser o caso que o algoritmo faz mais comparações de *hashes* e é onde temos probabilisticamente mais casamentos.

Tabela 5.5: Resultados da nossa proposta (ms).

		$m = 10$	$m = 15$	$m = 20$	$m = 25$	$m = 30$
Sequencial	$k = 1$	1685,71	1730,09	1868,27	1843,04	1871,96
	$k = 4$	1704,31	1748,13	1874,91	1877,79	1854,88
	$k = 16$	1711,93	1740,71	1893,02	1880,60	1881,48
	$k = 64$	1845,50	1770,15	1883,84	1902,18	1869,63
	$k = 256$	2441,45	1797,23	1968,87	1933,71	1931,52
OpenMP	$k = 1$	50,50	52,73	53,97	53,99	54,00
	$k = 4$	52,55	53,21	54,45	54,63	54,39
	$k = 16$	59,11	52,33	53,72	53,99	53,58
	$k = 64$	65,70	51,48	54,70	53,47	54,03
	$k = 256$	78,31	66,42	65,89	67,18	65,26
GPU	$k = 1$	3,58	4,11	5,08	5,11	5,05
	$k = 4$	3,75	4,05	5,22	5,07	5,07
	$k = 16$	4,33	4,06	5,15	5,12	5,10
	$k = 64$	4,90	4,17	5,16	5,11	5,12
	$k = 256$	5,43	4,70	5,25	5,26	5,27

Notamos o ganho com as diferentes formas de processamento pela Tabela 5.5. Quando comparamos a versão em OpenMP com a sequencial temos um ganho entre 25 e 35 vezes. Como esperado o ganho não varia muito com a mudança de parâmetros, quando analisamos os ganhos da GPU, temos um ganho entre 358 e 471 vezes em relação a versão sequencial e um ganho entre 10,42 e 14,42 em relação a versão em OpenMP. Para os dois casos o maior ganho está na situação em que temos probabilisticamente mais casamentos, em $m = 10$ e $k = 256$.

A Tabela 5.7 mostra o tempo de execução para $m = 10$ decomposta em passos, vemos que o tempo do Passo 2 é o menor em relação aos outros, na versão sequencial o passo 5 que é da comparação efetiva dos padrões é o mais demorado, nas demais implementações

Tabela 5.6: Ganhos da nossa proposta comparando as 3 implementações.

		$m = 10$	$m = 15$	$m = 20$	$m = 25$	$m = 30$
OpenMP vs Sequencial	$k = 1$	33,38	32,81	34,62	34,14	34,67
	$k = 4$	32,44	32,85	34,43	34,37	34,10
	$k = 16$	28,96	33,26	35,24	34,83	35,12
	$k = 64$	28,09	34,39	34,44	35,57	34,60
	$k = 256$	31,18	27,06	29,88	28,79	29,60
GPU vs Sequencial	$k = 1$	471,37	421,25	368,00	360,82	370,74
	$k = 4$	453,99	431,44	358,85	370,37	365,65
	$k = 16$	395,54	428,24	367,40	367,39	369,05
	$k = 64$	376,54	424,29	365,21	372,13	365,30
	$k = 256$	449,68	382,45	375,04	367,70	366,41
GPU vs OpenMP	$k = 1$	14,12	12,84	10,63	10,57	10,69
	$k = 4$	14,00	13,13	10,42	10,77	10,72
	$k = 16$	13,66	12,87	10,43	10,55	10,51
	$k = 64$	13,41	12,34	10,60	10,46	10,56
	$k = 256$	14,42	14,13	12,55	12,77	12,38

os tempos de execução dos passos ficam mais equilibrados. Quando comparamos a versão em OpenMP comparada a sequencial podemos ver ganhos próximos de 15, 30 e 60 para os Passos 3, 4 e 5 respectivamente, note que o maior ganho é do Passo em que efetivamente fazemos a comparação dos *hashes* e caracteres e o passo que demorava mais tempo na versão sequencial. Quando comparamos a versão em GPU com a OpenMP temos um ganho próximo a 17, 15 e 10 nos Passos 3, 4 e 5 respectivamente.

Tabela 5.7: Resultados da nossa proposta dividido em passos para $m = 10$ (ms)

		Passo 2	Passo 3	Passo 4	Passo 5	Total
Sequencial	$k = 1$	0,00	303,13	546,22	836,36	1685,71
	$k = 4$	0,00	303,47	547,01	853,83	1704,31
	$k = 16$	0,00	302,34	547,60	861,99	1711,93
	$k = 64$	0,00	303,76	547,55	994,18	1845,50
	$k = 256$	0,01	302,20	546,28	1592,96	2441,45
OpenMP	$k = 1$	0,02	20,20	16,92	13,36	50,50
	$k = 4$	0,20	20,68	17,72	13,94	52,55
	$k = 16$	0,61	19,84	17,71	20,94	59,11
	$k = 64$	1,71	18,41	18,40	27,18	65,70
	$k = 256$	5,94	20,30	23,30	28,78	78,31
GPU	$k = 1$	0,08	1,17	1,20	1,13	3,58
	$k = 4$	0,07	1,18	1,18	1,32	3,75
	$k = 16$	0,10	1,18	1,18	1,87	4,33
	$k = 64$	0,08	1,17	1,17	2,48	4,90
	$k = 256$	0,08	1,18	1,16	3,01	5,43

A Tabela 5.8 mostra o ganho da nossa proposta comparado com o Rabin-Karp paralelo para múltiplos padrões, a nossa proposta possui vários passos adicionais, se comparado ao Rabin-Karp tradicional, o intuito desses passos é melhorar o desempenho para múltiplos padrões. O que pode ser visto facilmente na tabela com o aumento de k. Tornar a comparação mais paralelizável utilizando a soma de prefixos, no caso sequencial era esperado que o tempo do algoritmo original fosse melhor para um k pequeno, visto que a *hash table* teria pouca ou nenhuma influência sobre os resultados e a soma de prefixos não ajudaria na paralelização visto que o algoritmo é sequencial. No caso do OpenMP foram utilizadas 128 threads, continuamos vendo o ganho com o aumento de k devido a *hash table*, mas verificamos uma piora no ganho quando checamos com a versão sequencial, isso ocorre devido a soma de prefixos gastar mais tempo de processamento que o ganho efetivo no tempo total de processamento. No caso da GPU, onde utilizamos milhares de *threads* verificamos um tempo menor de processamento em todos os casos, ou seja, mesmo sem a *hash table* a soma de prefixos gera um ganho no tempo total de processamento na verificação de casamentos e com a *hash table* esse ganho chega a ser 10 vezes maior para $k = 256$.

Tabela 5.8: Ganhos da nossa proposta nas 3 implementações comparados ao Rabin-Karp

		$m = 10$	$m = 15$	$m = 20$	$m = 25$	$m = 30$
Sequencial	$k = 1$	0,42	0,41	0,38	0,39	0,38
	$k = 4$	0,42	0,40	0,38	0,38	0,39
	$k = 16$	1,10	1,04	0,96	0,97	0,99
	$k = 64$	3,46	3,55	2,25	2,22	3,48
	$k = 256$	7,11	9,57	8,59	8,73	10,10
OpenMP	$k = 1$	0,28	0,25	0,24	0,27	0,24
	$k = 4$	0,43	0,38	0,31	0,26	0,34
	$k = 16$	0,60	0,75	0,55	0,53	0,56
	$k = 64$	1,61	2,01	1,87	1,88	1,88
	$k = 256$	5,36	6,32	5,48	5,47	5,51
GPU	$k = 1$	2,55	2,31	1,98	2,25	2,14
	$k = 4$	2,76	2,67	2,20	2,36	2,43
	$k = 16$	2,63	2,87	2,39	2,49	2,64
	$k = 64$	3,68	3,65	2,96	3,04	3,12
	$k = 256$	9,68	10,12	8,48	8,74	8,81

Capítulo 6

Conclusõão e Trabalhos Futuros

Nesta dissertação de mestrado foi investigado o problema da busca de padrões em texto e em especial na tarefa de casamento de padrões aproximado e exato. Após buscar implementações nessa área foi procurado evoluir e aprimorar as técnicas e resultados existentes. O texto aborda diversos aspectos da GPU, como sua evolução, seus componentes e o uso de memória, também é feito uma revisão do estado da arte em casamento de padrões, onde são apresentados diversos artigos e algoritmos sobre o tema.

Em uma primeira proposta foi utilizada instruções `shuffle` para acelerar a o processamento em de um casamento de padrões aproximado, além dessas instruções foi utilizado uma técnica para otimizar o acesso a memória compartilhada resultando em ganhos entre 1,31 e 1,84 comparado a outro algoritmo que não faz uso desta técnica.

Este trabalho apresentou uma proposta para o casamento de padrões exato a partir do Rabin-Karp. Para isso diversas implementações foram feitas, desde implementações sequenciais de forma monoprocessada, passando por vários núcleos de CPU sendo executados em paralelo utilizando OpenMP e por fim a execução da nossa proposta GPU. Com isso conseguimos comparar o desempenho do Rabin-Karp modificado para múltiplos padrões e adaptado para GPU com nossa proposta.

Esse trabalho foi importante pois o uso da soma de prefixos trouxe melhoria para a GPU em um problema muito discutido na literatura que é o casamento de padrões, melhorando a eficiência para o já conhecido Rabin-Karp. O ganho da nossa proposta chegou a ser 2 vezes mais rápido para um padrão e 10 vezes mais rápido para 256 padrões.

As técnicas utilizadas e apresentadas neste trabalho apresentaram um ganho considerável quando comparado as outras alternativas, como trabalhos futuros podemos explorar estas técnicas para a solução de problemas relacionados.

Referências

[1] Hwu, Wen mei W.: *GPU Computing Gems Emerald Edition*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1ª edição, 2011, ISBN 0123849888, 9780123849885. 1

[2] Man, D., K. Nakano e Y. Ito: *The approximate string matching on the hierarchical memory machine, with performance evaluation*. Em *Embedded Multicore Socs (MC-SoC), 2013 IEEE 7th International Symposium on*, páginas 79–84, Sept 2013. 1, 25, 28, 30, 31

[3] Munekawa, Y., F. Ino e K. Hagihara: *Design and implementation of the smith-waterman algorithm on the cuda-compatible gpu*. Em *BioInformatics and BioEngineering, 2008. BIBE 2008. 8th IEEE International Conference on*, páginas 1–6, Oct 2008. 1

[4] Ukkonen, E.: *Algorithms for approximate string matching*. Information and Control, vol. 64, no. 1-3, pp. 100-118, 1985. 1

[5] Valerievich, B. A., P. T. Anatolievna, B. M. Alekseevna e S. S. Vladimirovich: *The implementation on cuda platform parallel algorithms sort the data*. Em *2017 6th Mediterranean Conference on Embedded Computing (MECO)*, páginas 1–4, June 2017. 1

[6] Faujdar, N. e S. P. Ghrera: *A practical approach of gpu bubble sort with cuda hardware*. Em *2017 7th International Conference on Cloud Computing, Data Science Engineering - Confluence*, páginas 7–12, Jan 2017. 1

[7] Yong, K. K., M. W. Chua e W. K. Ho: *Cuda lossless data compression algorithms: A comparative study*. Em *2016 IEEE Conference on Open Systems (ICOS)*, páginas 7–12, Oct 2016. 1

[8] Lin, C. H., J. C. Liu e C. C. Li: *Speeding up rsa encryption using gpu parallelization*. Em *2014 5th International Conference on Intelligent Systems, Modelling and Simulation*, páginas 529–533, Jan 2014. 1

[9] Chen, B., B. Chen, H. Liu e X. Zhang: *A fast parallel genetic algorithm for graph coloring problem based on cuda*. Em *2015 International Conference on Cyber-Enabled Distributed Computing and Knowledge Discovery*, páginas 145–148, Sept 2015. 1

[10] NVIDIA Corporation: *NVIDIA CUDA C Programming guide*. `https://developer.nvidia.com/cudnn`, 2017. 1

[11] SANDES, Edans Flávius de Oliveira: *Comparação paralela de sequências biológicas longas utilizando unidades de processamento gráfico (gpus).* Tese de Mestrado, Universidade de Brasília, 2011. 1

[12] Cheng, Lok Lam, D.W. Cheung e Siu Ming Yiu: *Approximate string matching in DNA sequences.* Em *Proceedings of the Eighth International Conference on Database Systems for Advanced Applications*, páginas 303–310, March 2003. 1, 19, 25

[13] SANDES, Edans Flávius de Oliveira: *Algoritmos Paralelos Exatos e Otimizações para Alinhamento de Sequências Biológicas Longas em Plataformas de Alto Desempenho.* Tese de Doutoramento, Universidade de Brasília, 2015. 1

[14] Hains, D., Z. Cashero, M. Ottenberg, W. Bohm e S. Rajopadhye: *Improving cudasw++, a parallelization of smith-waterman for cuda enabled devices.* Em *2011 IEEE International Symposium on Parallel and Distributed Processing Workshops and Phd Forum*, páginas 490–501, May 2011. 1

[15] Wang, J., X. Xie e J. Cong: *Communication optimization on gpu: A case study of sequence alignment algorithms.* Em *2017 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, páginas 72–81, May 2017. 1

[16] *Kepler gk110 architecture whitepaper.* `https://www.nvidia.com/content/PDF/kepler/NVIDIA-Kepler-GK110-Architecture-Whitepaper.pdf`, 2012. 1, 7

[17] Nunes, L. S. N.: *Proposta e avaliação de um mecanismo para acelerar a busca em cadeias de caracteres utilizando gpgpu*, 2015. 1

[18] Nunes, L. S. N., J. L. Bordim, K. Nakano e Y. Ito: *A fast approximate string matching algorithm on gpu.* Em *2015 Third International Symposium on Computing and Networking (CANDAR)*, páginas 188–192, Dec 2015. 1, 2, 35

[19] Nunes, L. S. N., J. L. Bordim, K. Nakano e Y. Ito: *A memory-access-efficient implementation of the approximate string matching algorithm on gpu.* Em *2016 Fourth International Symposium on Computing and Networking (CANDAR)*, páginas 483–489, Nov 2016. 1, 2, 35

[20] NUNES, Lucas Saad Nogueira, Jacir Luiz BORDIM, Yasuaki ITO e Koji NAKANO: *A memory-access-efficient implementation for computing the approximate string matching algorithm on gpus.* IEICE Transactions on Information and Systems, E99.D(12):2995–3003, 2016. 1, 2

[21] *Gp100 pascal whitepaper - nvidia.* `https://images.nvidia.com/content/pdf/tesla/whitepaper/pascal-architecture-whitepaper.pdf`, 2016. 4, 8

[22] NVIDIA Corporation: *NVIDIA CUDA C Programming guide.* `http://docs.nvidia.com/cuda/cuda-c-programming-guide/`, 2017. 4, 5, 6, 7, 10, 17, 31

[23] Corporation, NVIDIA: *NVIDIA CUDA C best practice guide version 6.5.* `http://docs.nvidia.com/cuda/cuda-c-programming-guide/`, 2017. 6, 8

[24] Corporation, NVIDIA: *Nvidia's next generation cuda compute architecture: Fermi*. http://goo.gl/6PCeDu, 2009. 7

[25] *Nvidia geforce gtx 980*. https://international.download.nvidia.com/geforce-com/international/pdfs/GeForce_GTX_980_Whitepaper_FINAL.PDF, 2014. 8

[26] *Nvidia tesla v100 gpu architecture*. http://www.nvidia.com/object/volta-architecture-whitepaper.html, 2012. 8

[27] Flynn, Michael J.: *Some computer organizations and their effectiveness*. IEEE Transactions on Computers, C-21:948–960, 1972. 10

[28] Okamoto, S., Y. Ito, K. Nakano e J.L. Bordim: *Thorough evaluation of gpu shared memory load and store instructions*. Em *Computing and Networking (CANDAR), 2014 Second International Symposium on*, páginas 614–616, Dec 2014. 11, 13, 14, 15

[29] NVIDIA: *Inline ptx assembly in cuda*. http://docs.nvidia.com/cuda/inline-ptx-assembly, Junho 2015. 13

[30] NVIDIA Corporation: *GeForce GTX-960*. http://www.geforce.com/hardware/desktop-gpus/geforce-gtx-960, April 2017. 15, 38

[31] NVIDIA Corporation: *NVIDIA Visual Profiler*. https://developer.nvidia.com/nvidia-visual-profiler, May 2016. 16, 39

[32] Karp, Richard M. e M.O. Rabin: *Efficient randomized pattern-matching algorithms*. IBM Journal of Research and Development, 31(2):249–260, March 1987, ISSN 0018-8646. 19, 21

[33] Knuth D.E., MORRIS (Jr) J.H., Pratt V.R: *Fast pattern matching in strings*. SIAM Journal on Computing, 1977. 19, 23

[34] Porat, B. e E. Porat: *Exact and approximate pattern matching in the streaming model*. Em *2009 50th Annual IEEE Symposium on Foundations of Computer Science*, páginas 315–323, Oct 2009. 19

[35] Cormen, Thomas H., Clifford Stein, Ronald L. Rivest e Charles E. Leiserson: *Introduction to Algorithms*. McGraw-Hill Higher Education, 2nd edição, 2001, ISBN 0070131511. 19, 20, 21, 22, 23, 24, 25, 26

[36] Nakano, K.: *Efficient implementations of the approximate string matching on the memory machine models*. Em *Proceedings of the Third International Conference on Networking and Computing (ICNC 2012)*, páginas 233–239, Dec 2012. 26

[37] Noyes, J.: *Qwerty-the immortal keyboard*. Computing Control Engineering Journal, 9(3):117–122, June 1998, ISSN 0956-3385. 28

[38] Nakano, K.: *Simple memory machine models for gpus*. Em *Parallel and Distributed Processing Symposium Workshops PhD Forum (IPDPSW), 2012 IEEE 26th International*, páginas 794–803, May 2012. 28

[39] Nakano, K.: *The hierarchical memory machine model for gpus.* Em *Parallel and Distributed Processing Symposium Workshops PhD Forum (IPDPSW), 2013 IEEE 27th International*, páginas 591–600, May 2013. 28

[40] NVIDIA Corporation: *GeForce GTX-580.* `http://www.geforce.com/hardware/desktop-gpus/geforce-gtx-580`, Junho 2017. 31

[41] *Open multi-processing (openmp)*, 2018. `https://www.openmp.org/`. 41

[42] NVIDIA Corporation: *GeForce GTX-960.* https://www.nvidia.com/en-us/data-center/tesla-v100/, Jun 2018. 42

[43] Merrill, Duane: *A library of warp-wide, block-wide, and device-wide gpu parallel primitives.*, 2018. `https://nvlabs.github.io/cub/`. 49, 50, 52

[44] Merrill, Duane e Michael Garland: *Single-pass parallel prefix scan with decoupled look-back*, Março 2016. 49, 50, 51