Felipe Piñeiro Bitencourt

Uma solução para ordenar conjuntos de dados *string* com alta repetição de prefixos longos em GPU

Salvador, Bahia

Dezembro, 2013

Felipe Piñeiro Bitencourt

Uma solução para ordenar conjuntos de dados *string* com alta repetição de prefixos longos em GPU

Monografia apresentada à Banca Examinadora como requisito para obtenção do título de bacharel em Sistemas de Informação pela Universidade do Estado da Bahia

UNIVERSIDADE DO ESTADO DA BAHIA – UNEB
DEPARTAMENTO DE CIÊNCIAS EXATAS E DA TERRA – Campus I
COLEGIADO DE SISTEMAS DE INFORMAÇÃO

Orientador: Cláudio Alves de Amorim

Salvador, Bahia
Dezembro, 2013

Felipe Piñeiro Bitencourt

Uma solução para ordenar conjuntos de dados *string* com alta repetição de prefixos longos em GPU

Monografia apresentada à Banca Examinadora como requisito para obtenção do título de bacharel em Sistemas de Informação pela Universidade do Estado da Bahia

Trabalho aprovado. Salvador, Bahia, 12 de dezembro de 2013:

Cláudio Alves de Amorim

Marcos Barreto

Murilo do Carmo Boratto

Salvador, Bahia
Dezembro, 2013

Agradecimentos

Os agradecimentos principais são direcionados ao meu pai Lucival Andrade Bitencourt, à minha mãe Gumersinda Concepcion Piñeiro Bitencourt e ao meu irmão Fernando Piñeiro Bitencourt, por todo o carinho que sempre me dedicaram.

Agradeço também em especial aos meus amigos Bruno Vinicius, Juliana Fajardini e Raul Abreu, que estiveram presentes ao longo da minha jornada. Vamos seguir em frente, vamos de mãos dadas! Ao meu orientador, Cláudio Amorim, dedico boa parte dos meus agradecimentos por todos os seus conselhos e atenção. Ao professor, Murilo Boratto, deixo também os meus agradecimentos.

Por fim, e não menos importante, agradeço a todos que contribuíram direta ou indiretamente e fizeram com que a realização deste trabalho fosse possível.

"Deus, concedei-me,
a serenidade para aceitar as coisas que eu não posso modificar,
coragem para modificar as coisas que posso, e
sabedoria para saber a diferença.
Vivendo um dia de cada vez,
desfrutando um momento por vez,
aceitando as dificuldades como o caminho da paz...
(Oração da Serenidade)

Resumo

A ordenação é um problema clássico da computação e a sua relevância levou à concepção de soluções para classificar dados em Unidades de Processamento Gráfico (GPU). A eficiência de algoritmos para ordenar dados de tamanho variável, como *strings*, é importante para muitas aplicações. Apesar disso, poucos trabalhos têm abordado este problema em GPU. As *strings* possuem características específicas que influenciam no desempenho da classificação. Prefixo longo comum é uma dessas características. Neste trabalho é proposta uma solução eficiente para ordenar *strings* com alta ocorrência de prefixos longos comuns em GPU. Os experimentos mostram que para ambientes com alta repetição de prefixos, o algoritmo obteve bons resultados, chegando a dobrar sua própria eficiência para o conjunto com maior repetição de *strings*.

Palavras-chaves: Ordenação, *strings*, prefixos longos comuns, GPU.

Abstract

Sorting is a classic problem of computing and its relevance led to the design of solutions to sort data in Graphics Processing Unit (GPU). The efficiency of algorithms for sorting variable length data, such as strings, is important for many applications. Nevertheless, few studies have addressed this problem in GPU. The strings have specific characteristics that influence the performance of classification. Long common prefixes is one of those features. This paper proposes an efficient solution to sort data strings with high occurrence of long common prefixes in GPU. Results show that, for scenarios with high occurrence of repetition of prefixes, the algorithm had good performance, even doubling its efficiency for some cases.

Key-words: Ordering, strings, longest common prefixes, GPU.

Lista de ilustrações

Lista de tabelas

Lista de algoritmos

Sumário

1 Introdução

"A atividade de colocar as coisas em ordem está presente na maioria das aplicações em que os objetos armazenados têm de ser pesquisados e recuperados, tais como dicionários, índices de livros, tabelas e arquivos" (ZIVIANI, 2010, p. 101). Muitos algoritmos de ordenação foram propostos e seus desempenhos foram analisados profundamente. Afinal, para muitas aplicações, a ordem na qual os dados se apresentam pode exercer uma grande influência no desempenho e na simplicidade dos algoritmos que os manipulam (DANTAS, 1997).

Por algum tempo, diversos métodos de ordenação sequenciais foram desenvolvidos e alguns deles se estabeleceram como clássicos na computação; *Merge sort, Quicksort, Radix sort*, entre outros. Como arquiteturas de computadores evoluem, há uma necessidade contínua de explorar técnicas de classificação eficiente em arquiteturas emergentes (SATISH; HARRIS; GARLAND, 2009). Na vanguarda desses *hardwares* emergentes estão as placas de unidades de processamento gráfico (GPUs). A GPU é um processador maciçamente *multi-thread* que tem grande capacidade de processar trechos de código em paralelo.

Existe uma variedade razoável de soluções, disponíveis na literatura, para ordenar dados em memória interna, projetadas para GPUs. Alguns algoritmos buscam otimizar o desempenho da classificação em função de características específicas dos dados; o tipo, a distribuição inicial e o tamanho dos dados, são as principais especificidades escolhidas. Grande parte destes algoritmos resolvem o problema de ordenação para dados inteiros de tamanho fixo.

Assim como a ordenação de outros tipos de dados, a classificação de *strings* é fundamental para uma série de aplicações, como por exemplo, para a construção de índice do banco de dados, para alguns algoritmos de ordenação de sufixo e para ferramentas *MapReduce* (BINGMANN; SANDERS, 2013).

Embora exista uma quantidade razoável de trabalhos de ordenação sequencial de cadeias de caracteres, existem poucos trabalhos, conhecidos na literatura, sobre ordenação de *strings* em GPU. Davidson et al. (2012) relata ser o primeiro trabalho conhecido a desenvolver um algoritmo estável que pode classificar dados do tipo *string*. Ao analisar o comportamento do seu método em dois conjuntos de textos diferentes, Davidson percebeu que o desempenho do seu algoritmo é menos eficiente em ambientes com palavras de prefixos comuns. Quanto mais *strings* com prefixos comuns e quanto mais longos foram estes prefixos, mais o desempenho do seu algoritmo foi prejudicado.

Sequências de caracteres com prefixos longos comuns estão presentes em alguns

cenários, dentro e fora da computação. Três destes cenários são: 1. Conforme dados vão sendo classificados, as etapas finais da ordenação de alguns algoritmos tendem a ter dados mais similares a serem comparados. 2. Conjunto de textos em que autores utilizam padrões de palavras pra iniciar ou desenvolver frases. 3. Conjunto de palavras de alfabetos que contêm poucas letras, por ex. As sequências de DNA que contém apenas 4 letras.

Este trabalho propôs desenvolver e avaliar uma solução eficiente, em GPU, para ordenar conjuntos de sequências de caracteres com alta repetição de prefixos longos. Para isso, este trabalho utilizou uma abordagem *MSD radix sort*, onde foi possível decompor o problema em tarefas independentes e explorar ao máximo o número de *threads* da GPU. Além disso, como toda abordagem baseada em distribuição, o *MSD Radix Sort* evita as comparações repetidas dos prefixos compartilhados pelas *strings*.

O trabalho está organizado da seguinte forma. O Capítulo 2 apresenta uma visão ampla do problema de ordenação e faz um breve aprofundamento sobre ordenação de strings, discorrendo sobre os principais aspectos e técnicas utilizadas. Um dos principal ambientes de alto desempenho e plano de fundo deste trabalho, A GPU é apresentada no Capítulo 3. O Capítulo 4 aborda os modelos utilizados e duas técnicas que exploram um alto nível de paralelismo para ordenação de dados em GPU. O Capítulo 5 descreve a solução proposta nesse trabalho para ordenação de strings LCP em GPU. O Capítulo 6 apresenta os resultados experimentais. Por fim, é feito a conclusão e são apresentadas as sugestões de trabalhos futuros.

Parte I

Fundamentação Teórica

2 Ordenação

Os algoritmos de ordenação são exemplos clássicos de como resolver problemas utilizando computadores. Ordenar corresponde ao processo de rearranjar um conjunto de objetos em ordem ascendente ou descendente (ZIVIANI, 2010). Encontrar um objeto em um universo de dados desordenados, é como "encontrar uma agulha no palheiro". Onde o objeto desejado representa a agulha e o resto do universo, as palhas. Listas telefônicas, dicionários, bibliotecas, grandes acervos, quanto maior a quantidade de elementos deste universo, maior o número de palhas e mais demorada será a tarefa de achar o objeto desejado (a agulha). É para facilitar a recuperação posterior de itens de um conjunto que os algoritmos de ordenação são projetados.

Segundo Knuth (1998), para que uma relação de ordem seja estabelecida, deve-se cumprir duas propriedades básicas:

1. Uma e somente uma das possibilidades $a < b$, $a = b$, ou $b < a$ pode ocorrer (princípio da tricotomia).

2. Se $a < b$ e $b < c$, então $a < c$ (princípio da transitividade).

O cumprimento dessas condições é indispensável para que a ordenação seja feita, e quando o critério utilizado as satisfaz, diz-se que uma relação de ordem linear ou total é estabelecida (DANTAS, 1997).

Os métodos de ordenação são classificados de diversas formas. Uma delas decorre da localidade dos dados, se todos os dados a serem ordenados cabem na memória principal, então o método de classificação é chamado ordenação interna. Quando o tamanho dos elementos excedem a capacidade da memória principal, e por isso, têm de ser armazenados em memória secundária, o método é classificado como um algoritmo de ordenação externa. Esses dois tipos de algoritmos exigem esforços de implementação diferentes. Afinal, enquanto algoritmos de ordenação interna podem acessar todos os dados facilmente, algoritmos de ordenação externa têm que contornar os altos custos de acessos à memória secundária.

Outra categorização comum dos algoritmos de classificação corresponde à técnica utilizada para a ordenação. A grande maioria dos métodos é baseada em comparações das chaves. Entretanto, existem métodos de ordenação que utilizam o princípio da distribuição (ZIVIANI, 2010).

2.1 Ordenação por Comparação

Um algoritmo de classificação baseado em comparação é um método de ordenação que estabelece a ordem dos elementos de um conjunto através de uma operação de comparação direta dos objetos. O Algoritmo 2.1 exemplifica com uma função de comparação de números inteiros.

Algoritmo 2.1: Comparação de dois números inteiros em C

```
1 int compareLess(int a, int b) {
2   if(a < b) {
3         return 1;
4   }
5   else {
6         return 0;
7   }
8 }
```

Existem diversos algoritmos de ordenação que utilizam o princípio da comparação de chaves; os mais conhecidos são: *insertion sort*, *selection sort*, *shellsort*, *quicksort*, *mergesort* e *heapsort*. Os primeiros, *insertion* e *selection sort*, são considerados métodos simples. Enquanto os outros são considerados métodos eficientes. Os métodos eficientes são os mais utilizados para uma ampla variedade de problemas. Porém, existe um grande número de situações em que é melhor usar os métodos simples do que usar os métodos mais sofisticados. Apesar de os métodos mais sofisticados usarem menos comparações, estas comparações são complexas nos detalhes, o que torna os métodos simples mais eficientes para pequenos arquivos. (ZIVIANI, 2010)

Algoritmos de ordenação por comparação podem classificar qualquer conjunto de dados especificando apenas a função de comparação entre dois elementos desse conjunto (DAVIDSON et al., 2012). Essa função pode ser facilmente adaptada para os mais variados tipos de dados, como é demonstrado pelo Algoritmo 2.2, onde o Algoritmo 2.1 é alterado para poder comparar sequências de caracteres.

Algoritmo 2.2: Comparação de duas sequências de caracteres em C

```
1 int compareLess(char *a, char *b) {
2   int i;
3   for(i=0 ;; i++) {
4         if((a[i] == '\0') || (a[i] < b[i])) {
5           return 1;
```

```
6           }
7           if((b[i] == '\0') || (a[i] > b[i])) {
8             return 0;
9           }
10    }
11  }
```

Embora algoritmos por comparação sejam mais flexíveis e apresentem bons resultados para variados problemas, algoritmos por distribuição são por vezes mais eficientes, e por isso também aparecem com frequência nas soluções e na literatura da Computação.

2.2 Ordenação por Distribuição

Ordenação por distribuição é a técnica utilizada por algoritmos de classificação em que os dados são distribuídos em várias estruturas intermediárias que são recolhidas e depois reorganizadas. Nesse caso, não existe comparação entre chaves, a ordenação é garantida pela natureza dos dados, que são distribuídas em estruturas intermediárias, e por sua vez, são recolhidas numa ordem pré-definida.

Knuth (1998) exemplifica a resolução de um algoritmo de ordenação por distribuição com o seguinte caso:

Considerado o problema de ordenar um baralho com 52 cartas não ordenadas. Primeiro é necessário estabelecer o critério de ordem. Uma ordem comum para cartas de baralho é; $A < 2 < 3 < \ldots < 10 < J < Q < K$ para os valores das cartas, e para os naipes; $\spadesuit < \heartsuit < \diamondsuit < \clubsuit$.

Uma carta deve preceder outra caso:

1. Seu naipe seja menor do que o outro naipe,
2. Seu naipe é igual ao outro naipe, mas seu valor é menor.

Assim, $\spadesuit A < \spadesuit 2 < \ldots < \heartsuit K < \diamondsuit A < \ldots < \clubsuit Q < \clubsuit K$.

Um solução em alto nível para este problema utilizando ordenação por distribuição pode ser dada da seguinte forma:

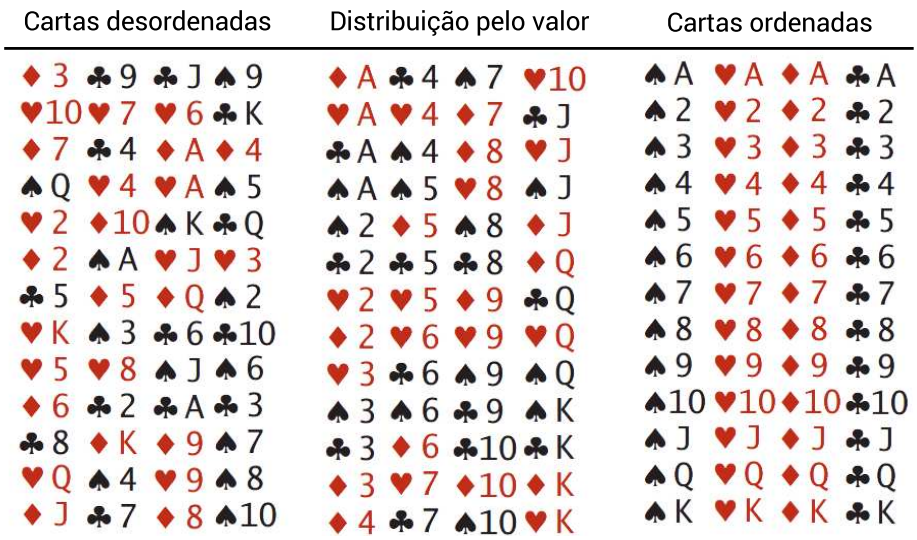
1. Distribuir as cartas em treze montes, colocando a carta em cada monte correspondente ao seu valor.
2. Coletar os montes na ordem de valores citada; $A < 2 < 3 < \ldots < Q < K$.

3. Distribuir novamente as cartas, mas agora, em quatro montes, colocando em monte todas as cartas de seus respectivos naipes.

4. Colete os montes respeitando a ordem citada; ♠ < ♡ < ♢ < ♣.

A Figura 1 ilustra os passos desse algoritmo.

Figura 1 – Ordenação de cartas. Fonte: Sedgewick e Wayne (2011).



Algoritmos de classificação de distribuição são muito utilizados em soluções paralelas, em que os subconjuntos individuais são classificados separadamente em diferentes processadores e depois recombinados.

2.3 Ordenação de Strings

As técnicas de ordenação permitem apresentar um conjunto amplo de algoritmos distintos para resolver uma mesma tarefa. Dependo da aplicação, cada algoritmo considerado possui uma vantagem particular sobre os outros (ZIVIANI, 2010). Para muitas aplicações de classificação, o tipo de dados ordenados são *strings*. *Strings* possuem propriedades específicas que devem ser exploradas para que algoritmos tenham um melhor desempenho.

Nesta seção serão descritas as propriedades particulares das *strings* e depois apresentadas as principais abordagens para ordená-las eficientemente.

2.3.1 Definição

String é uma sequência finita de n caracteres pertencentes a um alfabeto. Formalmente, a *string* é um símbolo,

$$S = s_1 s_2 \dots s_n,$$

onde n é um número inteiro positivo e cada s_i é um carácter de um determinado alfabeto. O comprimento da cadeia é determinado pelo inteiro n.

2.3.2 Alfabeto

Segundo Houaiss (1996), a definição de um alfabeto é descrita como o conjunto das letras de um sistema de escrita, dispostas em ordem convencionalmente estabelecida. Na computação, o alfabeto é o conjunto de caracteres possíveis para uma determinada *string*. Por exemplo, $\{a, b, \dots, z\}$ é o alfabeto de todas as letras minúsculas usadas na língua portuguesa.

Por muito tempo, os programadores restringiam a atenção para caracteres codificados em *ASCII* de 7 *bits* ou *ASCII* estendido de 8 *bits*, mas muitos aplicativos modernos usam *Unicode* de 16 *bits* (SEDGEWICK; WAYNE, 2011). Estes e outros alfabetos comuns em problemas resolvidos usando computação são mostrados na Figura 2.

Figura 2 – Alfabetos padrões. Fonte: Sedgewick e Wayne (2011).

name	size	characters
BINARY	2	01
DNA	4	ACTG
OCTAL	8	01234567
DECIMAL	10	0123456789
HEXADECIMAL	16	0123456789ABCDEF
PROTEIN	20	ACDEFGHIKLMNPQRSTVWY
LOWERCASE	26	abcdefghijklmnopqrstuvwxyz
UPPERCASE	26	ABCDEFGHIJKLMNOPQRSTUVWXYZ
BASE64	64	ABCDEFGHIJKLMNOPQRSTUVWXYZabcdef ghijklmnopqrstuvwxyz0123456789+/
ASCII	128	*ASCII characters*
EXTENDED_ASCII	256	*extended ASCII characters*
UNICODE16	65536	*Unicode characters*

Standard alphabets

2.3.3 Ordem Lexicográfica

Diferentemente da comparação de dados numéricos, onde a comparação se dá em uma única passagem, a comparação de dados do tipo *string* consiste em percorrer os caracteres em várias passagens. A ordem é estabelecida pela primeira diferença de caracteres entre as strings. Essa ordem é chamada de ordem lexicográfica.

Ordem lexicográfica é comum na vida cotidiana, como por exemplo, a ordem alfabética utilizada em dicionários e listas (FRüHWIRTH, 2005). É natural que em alguns

conjuntos, as sequências de caracteres tenham tamanhos variáveis. Para estabelecer uma relação de ordem nesses casos, o que geralmente é feito é preencher a sequência mais curta com símbolos que são considerados o elemento mínimo de um alfabeto.

Uma definição generalizada da ordem lexicográfica é:

Sendo, $A = a_1a_2 \ldots a_n$ e $B = b_1b_2 \ldots b_m$, duas sequências de caracteres. Então A é menor do que B, se e somente se, existe um número inteiro k não negativo com as seguintes propriedades:

1. $k \leq n$ e $k \leq m$;
2. Para todo inteiro positivo $i \leq k$, $a_i = b_i$;
3. Ou $a_{k+1} < b_{k+1}$ ou então $k = n$ e $k < m$

Logo, para o alfabeto da língua portuguesa,

$a < aa < aa \ldots a < ab < \ldots < az < aza < \ldots < z < \ldots < zz \ldots$

É importante ressaltar que a ordem lexicográfica também garante os princípios da tricotomia e da transitividade, citadas no começo do Capítulo 2, que estabelecem uma relação de ordem linear.

2.3.4 Complexidade

O aspecto predominante na escolha de um algoritmo de ordenação é o tempo gasto para ordenar um arquivo (ZIVIANI, 2010). Os algoritmos de classificação tradicionais, como *merge sort*, *quick sort*, *insertion sort*, medem a complexidade com base no número de comparações que são feitas. Esses algoritmos baseados em comparação leem os elementos da lista e determinam qual de dois elementos ocorre em primeiro lugar, mas no caso de *string*, cada caractere é analisado e o comprimento é um fator importante na medição da complexidade (ANGRISH; GARG, 2011).

Para chaves numéricas, $O(n \log n)$ comparações são suficientes, para os principais algoritmos eficientes, como *merge sort*. Mas para classificação de *strings*, esta medida simples de contar as comparações entre chaves não reflete adequadamente a complexidade dos algoritmos. Comparar duas sequências de caracteres leva tempo proporcional ao comprimento do prefixo mais longo comum +(mais) 1(um), uma vez que esta quantidade de caracteres têm de ser comparados a fim de resolver a comparação (SEIDEL, 2010).

No entanto, não é de todo claro quantas comparações ocorrem em uma execução de um algoritmo de classificação de *strings*. É inadequado considerar apenas a quantidade de *strings* como atributo para mensurar a complexidade. Pois, ao adicionar um prefixo comum para todas as n *strings* de um conjunto o tempo de execução aumenta enquanto

n permanece o mesmo. Considerar a soma de todos os comprimentos das *strings*, como outro parâmetro também não é adequado, pois, dessa forma, iguala-se o custo da adição de prefixos comuns para todas as *strings*, que aumenta o número de comparações de caracteres, com a adição de sufixos comuns a todas as *strings*, que não provoca comparações adicionais.

Esse problema de como parametrizar a complexidade dos algoritmos de classificação de string pode ser contornado, considerando d o tamanho do prefixo distintivo de um conjunto de dados, isto é, o número total de caracteres que precisam ser inspecionados, a fim de estabelecer a ordem lexicográfica. Portanto, para um algoritmo de ordenação baseada em comparações de caracteres, obter $O(d + n \log n)$ de complexidade, é o desejável (BINGMANN; SANDERS, 2013).

2.3.5 Prefixo Longo Comum

Para entender sobre prefixos longos comuns (Longest Common Prefix - LCP), primeiro é necessário definir o que são prefixos e prefixos comuns. Kakehi e Katsuhiko (2008), apresentam com bastante clareza essas definições.

Sendo p, s e t, três *strings* quaisquer

Definição 1 (Prefixo)**.** *p é um prefixo de s se $s = ps'$ para qualquer string s'.*

Por exemplo, uma *string* "*ab*" é prefixo de uma *string* "*abbba*". Vale lembrar, que qualquer *string* é um prefixo da própria sequência. No mesmo exemplo, a *string* "*abbaa*" é prefixo dela mesma.

Definição 2 (Prefixo Comum)**.** *p é um prefixo comum de s e t se $s = ps'$ e $t = pt'$ para quaisquer strings s' e t'.*

Por exemplo, uma *string* "*ab*" é prefixo comum das *strings* "*abbaa*" e "*aba*".

Definição 3 (Prefixo Longo Comum)**.** *Quando p é o mais longo entre os prefixos comuns de s e t, ou seja, o comprimento de p é maior que o comprimento de p' para qualquer prefixo comum $p' \mathrel{!}= p$ de s e t, então p é o maior prefixo comum (LCP) de s e t.*

Por exemplo, sendo duas *strings* "*abbaba*" e "*abbaa*" então as "*a*", "*ab*", "*abb*" e "*abba*" são os prefixos comuns, e "*abba*" é o LCP.

Sabendo qual é o LCP compartilhado por todas as *strings* de um conjunto, a ordenação pode ser redefinida da seguinte forma,

Sendo as strings $s = ps'_1 s'_2 \dots s'_n$, $t = pt'_1 t'_2 \dots t'_m$ e p o LCP de s e t,

1. $s > t$ se $s'_1 > t'_1$,

2. $s < t$ se $s'_1 < t'_1$,

3. $s = t$ se $s'_1 = t'_1$

Conjuntos de dados com alta ocorrência de prefixos longos comuns são recorrentes em diversos problemas, como em arrays de sufixos (KäRKKäINEN; MANZINI; PUGLISI, 2009), alfabetos com conjuntos de poucos caracteres, determinadas etapas de algoritmos de ordenação, entre outros (SINHA; ZOBEL, 2003).

2.3.6 Principais abordagens

Comparar os prefixos comuns repetidas vezes é custoso. Como visto anteriormente, o LCP compartilhado entre as *strings* é um fator que determina a complexidade da ordenação de *strings*. Conjuntos de dados com alta quantidade de prefixos longos comuns, quando comparados com conjuntos de poucos prefixos longos comuns, levam tempo maior. É por esta razão que os métodos destinados especificamente para ordenar *strings* podem ser substancialmente mais eficientes que algoritmos de ordenação de propósito geral (SINHA; ZOBEL, 2003).

2.3.6.1 Radix sort

Radix sort é um método de classificação baseado na técnica de distribuição dos dados. Ao invés de comparar os caracteres na sua totalidade, eles são decompostos em pedaços de uma sequência de tamanho fixo. Existem duas principais abordagens de *Radix sort*: dígito mais significativo (MSD) e dígito menos significativo (LSD).

2.3.6.1.1 LSD Radix sort

LSD Radix sort foi o método utilizado por máquinas antigas classificadoras de cartões perfurados que foram desenvolvidas antes de existirem os computadores eletrônicos (KNUTH, 1998). Para um conjunto de *strings* de comprimento w, *LSD Radix sort* faz w passagens começando do símbolo menos significativo até o mais significativo, para poder ordenar o conjunto por completo. Do ponto de vista teórico,*LSD Radix sort* é importante porque é um algoritmo de ordenação de tempo linear. Não importa quão grande seja a quantidade de dados, faz w passagens através dos dados (SEDGEWICK; WAYNE, 2011).

Os principais motivos para LSD não ser uma abordagem muito preterida para classificação de *strings*, é que ela é impraticável para cadeias de comprimento variável (SINHA; ZOBEL; RING, 2007) e necessita inspecionar todos os caracteres da entrada, o que é desnecessário em abordagens MSD (SEDGEWICK; WAYNE, 2011).

LSD Radix sort é um dos métodos usados para ordenar as cartas de um baralho. A Figura 1 representa uma ordenação feita com os passos deste algoritmo. Começa a clas-

sificação com o símbolo menos significativo (os valores das cartas) e termina a ordenação com a distribuição dos símbolos mais significativos (os naipes).

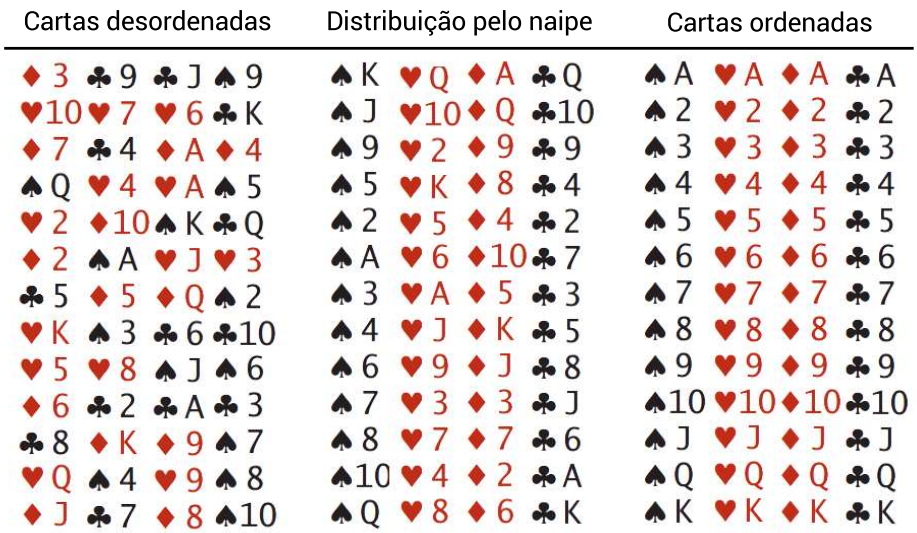
2.3.6.1.2 MSD Radix sort

Para implementar um classificador para problemas em que as strings não são necessariamente todas do mesmo tamanho, *MSD Radix sort* vai no sentido contrário do *LSD Radix sort*, partindo do símbolo mais significativo para o menos significativo, assim como a ordem lexicográfica sugere. A principal vantagem é que *MSD Radix sort* examina apenas os prefixos distintivos, uma abordagem atraente porque ele usa a quantidade mínima de informações necessárias para completar a ordenação. Portanto tem complexidade de tempo $O(d + n)$, onde d é o número total de caracteres dos prefixos distintivo e n a quantidade de *strings*.

O algoritmo segue o modelo padrão do *Radix sort*, a cada iteração coloca as *strings* em baldes de acordo com seus respectivos caracteres. Cada balde pode ser classificado de forma independente, então o algoritmo pode repetir os passos, distribuindo os dados em novos baldes ou usar um outro classificador para ordenar os dados do balde. Geralmente, quando a quantidade de elementos em um balde é pequeno, usa-se um algoritmo simples de classificação, como *insertion sort*. Independente se é chamado um outro algoritmo de ordenação ou um chamada recursiva do *MSD Radix sort*, os caracteres que já foram examinados são desconsiderados, pois os elementos pertencentes ao mesmo balde compartilham de um mesmo prefixo.

A Figura 3 é uma variação de ordenação de cartas que segue o modelo *MSD Radix sort*. Só que neste caso, como esperado, a ordenação é iniciada pelo símbolo mais significativo, o naipe.

Figura 3 – Ordenação de cartas começando pelo naipe. Fonte: Sedgewick e Wayne (2011).



Os custos no princípio de *MSD Radix sorts* se aproximam do mínimo teórico, lendo apenas o prefixo distintivo e só a leitura de cada caracter no prefixo uma vez. Um dos problemas de *MSD Radix sort* é a sensibilidade em relação à distribuição dos dados (SEDGEWICK; WAYNE, 2011).

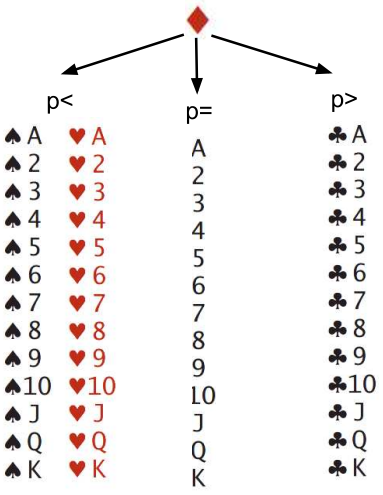
1. Para dados de conjuntos aleatórios, *MSD Radix sort* examina apenas caracteres suficientes para classificar os dados e, o tempo de execução é sublinear ao número de caracteres de dados.

2. Para dados de conjuntos não aleatórios, *MSD Radix sort* pode precisar examinar mais caracteres que no caso das entradas aleatórias, dependendo dos dados. No pior caso, *MSD Radix sort* tem que examinar todos os símbolos dos dados, de modo que o tempo de execução é linear ao número total de símbolos do conjunto (como o *LSD Radix sort*). O pior caso ocorre quando os dados de um conjunto são todos iguais.

2.3.6.2 Multikey quicksort

Multikey quicksort é um híbrido de *quicksort* e *MSD Radix sort*. Como em padrão de um *quicksort*, um pivô é selecionado (aleatoriamente, mediana, etc), então os dados são distribuídos em três partições ($p <$, $p =$, e $p >$) com base no(s) primeiro(s) caractere(s) do pivô. Todos os os elementos menores são distribuídos ao balde $p <$, os de valores iguais ao balde $p =$, e os de valores maior no balde $p >$. Para o balde $p =$, o próximo caractere das sequências é comparado, afinal todas as outras chaves desse balde compartilham o mesmo caractere analisado anteriormente. As strings nas outras duas partições são divididas repetindo o processo anterior para cada balde. Assim como em *MSD Radix sort*, para baldes de tamanhos pequenos, geralmente é aplicado um algoritmo simples de ordenação para concluir a classificação.

Figura 4 – Primeiro passo de uma ordenação de cartas feita por um multikey quicksort. Fonte: Elaborada pelo Autor.



Essa abordagem também evita a principal desvantagem de muitos algoritmos de ordenação para *strings*, ou seja, as comparações desnecessárias de prefixos comuns dos dados. Porém um caractere pode ser inspecionado várias vezes, até que ele esteja em um balde $p =$ ao pivô da partição (SINHA; ZOBEL, 2004). A escolha do pivô pode determinar a quantidade de inspeções repetidas aos caracteres dos baldes $p <$ e $p >$. Como em *quick sort*, a escolha de um pivô ruim pode gerar o pior caso.

Por ser uma abordagem de entendimento mais complexo, dificilmente pessoas escolhem *Multikey quicksort* para ordenar um baralho. Mas, caso ele seja escolhido, um possível primeiro passo dele é ilustrado na Figura 4. Percebe-se que o primeiro pivô possui o naipe de ♢. Logo, todas as cartas dos naipes que precedem ♢ são colocadas em uma estrutura $p <$. As cartas de ♣, sucedem às de ♢ e por isso, são colocadas em $p >$. As cartas de $p =$ são armazenadas apenas com seus respectivos valores, pois já é sabido que todas elas compartilham o mesmo naipe (♢).

3 Computação de Alto Desempenho com GPU

A Computação de Alto Desempenho é uma subárea da Ciência da Computação, que surgiu entre os anos de 1960 e 1970 com o desenvolvimento de supercomputadores. Embora trabalhos desenvolvidos por von Neumann na década de 1940 já discutissem a possibilidade de algoritmos paralelos para a solução de equações diferenciais (FILHO, 2007), é com o surgimento desses supercomputadores que ocorre a inovação das arquiteturas de computadores, fomentando o pensamento de quebrar um problema em subproblemas menores para resolvê-los simultaneamente. A partir de então, o paralelismo começa a ser um caminho viável e eficiente para solucionar problemas computacionais.

Segundo Nobre (2011), as principais motivações para a utilização da computação de alto desempenho são:

1. Resolver problemas de forma mais rápida. Fundamental para aplicações que necessitam de resposta rápida, como, por exemplo, previsão do tempo, aviso de tempestade, aviso de tsunami, etc.

2. Resolver mais casos do mesmo problema. Isso é importante para qualquer problema que necessita expandir a quantidade de dados processados e não pode sofrer aumento no tempo de resolução.

3. Maior poder computacional. São os casos dos problemas que precisam resolver problemas maiores e de uma maneira ainda mais rápida. Geralmente são problemas mais complexos como modelagem de fenômenos físicos do mundo real.

Atualmente, a Unidade de Processamento Gráfico (GPU) é destacada como uma plataforma computacional poderosa. Uma variedade de trabalhos usando GPU para acelerar a resolução de problemas científicos (VIEIRA; CAMPOI, 2009), (LIMA, 2012), (NOBRE, 2011), (HEIMLICH, 2009), (SAYURI; NAKASHIMA, 2013), mostraram que ganhos de desempenho significativos podem ser alcançados através do uso da GPU para problemas computacionais de dados paralelos. Para alcançar estes aumentos de velocidade, é necessário entender a arquitetura e o modelo de programação para uma GPU.

3.1 Paralelismo de dados

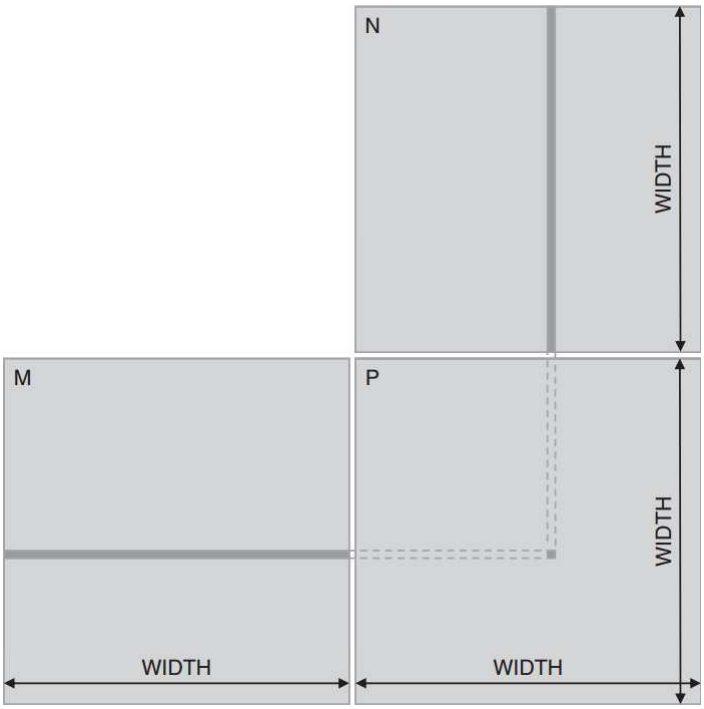
Uma questão importante na elaboração de um algoritmo paralelo para um dado problema é a maneira pela qual a carga de processamento é dividida entre os vários

processadores (NOBRE, 2011).

Paralelismo de dados é uma técnica aplicada em problemas onde os dados envolvidos são distribuídos e processados independentemente em diferentes nós de um ambiente paralelo para serem executados por uma mesma instrução de maneira simultânea. Em uma máquina de múltiplos núcleos, por exemplo, o paralelismo de dados ocorre quando cada processador executa a mesma tarefa em diferentes partes dos dados que foram distribuídos a ele.

Um exemplo básico onde o paralelismo de dados é aplicável é a multiplicação de matrizes. Cada elemento da matriz resultante pode ser calculado independentemente. Afinal, cada elemento é gerado pelo produto de uma linha de uma matriz pela coluna de outra. Essa independência é o que caracteriza o paralelismo dos dados e permite o processamento paralelo, onde os dados das matrizes produtos podem ser distribuídos e cada novo elemento pode ser calculado simultaneamente. A Figura 5 ilustra o cálculo de um ponto resultante da multiplicação de duas matrizes.

Figura 5 – Multiplicação de matrizes. Fonte: Kirk (2011).



GPUs são muito bem projetadas para resolver eficientemente problemas que possibilitem o uso de paralelismo de dados. Afinal, ela é composta de dezenas ou mesmo centenas de componentes de computação paralela que processam todos os dados independentemente (BEETS, 2013).

3.2 GPU

Criada em 1997 pela NVIDIA, a GPU foi projeta inicialmente para gerar os gráficos que seriam mostrados nos monitores, retirando essa carga do processador (NOBRE, 2011). Entretanto, a alta capacidade de processar trechos de código em paralelo, despertou o interesse de diversos desenvolvedores e pesquisadores que desejavam obter soluções mais eficientes para alguns problemas (LIMA, 2012).

Para simplificar o desenvolvimento de códigos para GPU, em áreas que não fossem de processamento de imagens, algumas ferramentas foram desenvolvidas. É nesse contexto que surgiu a definição de programação de propósito geral em GPUs (General-purpose computing on graphics processing units - GPGPU). A partir de então, algumas linguagens são criadas especificamente para apoiar o desenvolvimento de códigos paralelos em processadores gráficos, dentre as quais podemos citar CUDA, que sera detalhada na Seção 3.3.

As GPUs deixaram de ser apenas um poderoso *hardware* gráfico, para se tornar um poderoso processador altamente paralelo que suporta programação. Devido à alta necessidade de resolver problemas complexos em um tempo viável para vários campos de pesquisas, biológicas, matemáticas, geográficas, etc, as GPU's têm sido utilizada por muitos pesquisadores e desenvolvedores para aproveitar o seu poder de processamento paralelo para computação científica e de propósito geral (OWENS et al., 2007).

3.2.1 Arquitetura de uma GPU

Enquanto a CPU é otimizada para o desempenho de código sequencial, a GPU é projetada como mecanismo sofisticado de cálculo numérico (KIRK, 2011). Essa diferença é observada pela distribuição dos componentes de Unidade de Controle (UC) e Unidade Lógica e Aritmética (ULA) nas arquiteturas de ambas plataformas. A Figura 6 ilustra o projeto dessas duas arquiteturas. Por priorizar componentes de cálculos ao invés de unidades de controle, A GPU impõe uma grande vantagem à CPU para resolver problemas computacionais paralelos.

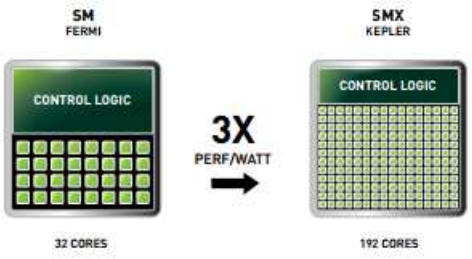
Figura 6 – Diferença de projeto das arquiteturas CPU e GPU. Fonte: NVIDIA (2013).

A GPU é organizada em uma matriz de multiprocessadores de *streaming* (SMs). Cada SM contém uma quantidade de processadores de *streaming* (SPs) que compartilham

a lógica de controle e cache de instruções. Cada SP possui pipelines completos de operações aritmética e de ponto flutuante. Toda GPU tem uma memória global, atualmente são memórias DRAM's e chegam a ter capacidade superior a 8 gigabytes. A memória global tem alta latência de acesso, porém existem na GPU outros tipos de memórias que são de baixa latência. A Seção 3.3.2 especifica estes outros tipos de memória.

A arquitetura Kleper é a arquitetura de GPU mais atual disponível. A GeForce GTX 680 e a GeForce GT640M foram as primeiras placas a utilizar essa tecnologia. A principal evolução da arquitetura Kleper foi o redesign da SM para obter maior desempenho e eficiência energética. Esse novo modelo de SM, chamado de SMX, possibilitou a construção de uma placa de vídeo de 192 SP's por SM, totalizando 1536 SPs no chip. A Figura 7 apresenta o *design* dos multiprocessadores da arquitetura Kleper e o da sua antecessora, a arquitetura Fermi.

Figura 7 – Comparação dos multiprocessadores das arquiteturas Fermi e Kleper. Fonte: NVIDIA (2012).



3.3 CUDA

A arquitetura CUDA é uma plataforma de computação projetada pela NVIDIA para facilitar o desenvolvimento de programas que explorem a alta capacidade de computação paralela das GPUs. CUDA fornece aos desenvolvedores uma extensão da linguagem de programação C e um conjunto poderoso de APIs abstraindo os detalhes de hardware.

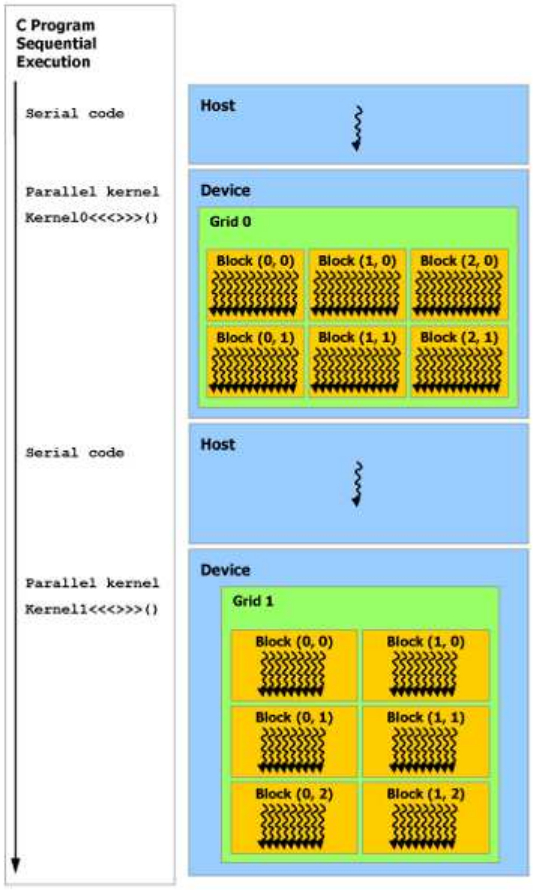
Com todo esse suporte de desenvolvimento, inúmeros desenvolvedores estão utilizando CUDA para resolver problemas em aplicativos comerciais e domésticos (VIEIRA; CAMPOI, 2009). Por trás dessa facilidade de desenvolvimento e popularidade de mercado, existe um modelo de programação e hierarquia de memória muito bem estruturados que serão detalhados nas próximas seções.

3.3.1 Modelo de programação em CUDA

No modelo de programação CUDA, um programa pode ser dividido em duas partes: o código *host* que é executado na CPU, e o código *device*, que é executado na GPU. Toda função definida para executar em uma placa gráfica é denominada de *kernel*. As funções

kernel devem gerar um grande número de *threads* para explorar o paralelismo de dados (KIRK, 2011). A Figura 8 ilustra a execução de um programa CUDA.

Figura 8 – Execução de um programa CUDA. Fonte: NVIDIA (2013).

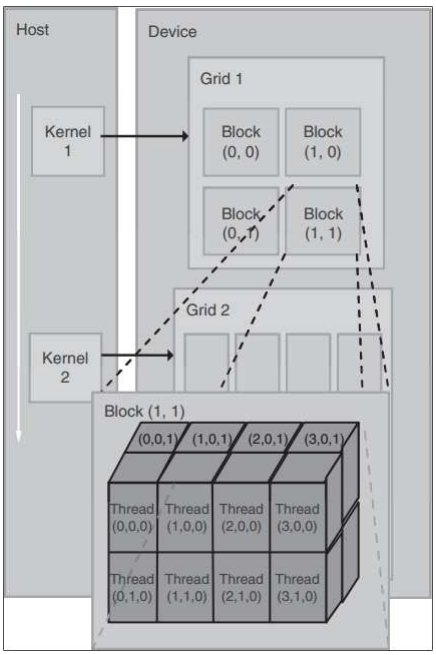


CUDA tem uma hierarquia composta de *threads*, blocos e *grids*. Quando um *kernel* é disparado, ele é executado como uma grade de *threads* paralelas. Cada *grid* normalmente é composto por milhares de blocos, que por sua vez é composto por até algumas milhares de *threads*. Para grande parte das GPUs NVIDIA, os blocos de um *grid* podem ser organizados uni ou bidimensionalmente, placas de vídeo modernas já aceitam que eles sejam definidos tridimensionalmente. As *threads* de um bloco também podem ser organizadas como uma matriz uni, bi ou tridimensional. A Figura 9 ilustra uma suposta organização de poucas *threads* em CUDA.

Threads e blocos possuem identificadores únicos dentro do seu bloco e seu *grid*, respectivamente. Através destas variáveis é possível manipular *threads* e blocos individualmente, isto é muito útil para garantir o processamento individual de uma grande quantidade de dados.

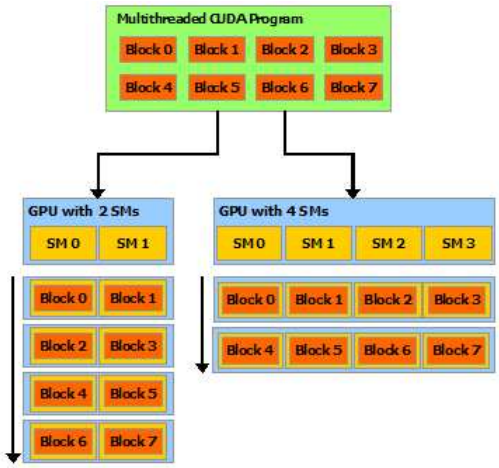
Por fim, os blocos são escalonados em vários multiprocessadores. À medida que a execução de um bloco termina, um novo bloco é alocado àquele processador até que todos os blocos tenham sido executados. É importante ressaltar que a ordem de execução dos

Figura 9 – Organização hierárquica de *threads* em CUDA. Fonte: Kirk (2011).



blocos não pode ser determinada. O *hardware* é responsável por isso através de gerenciamentos internos de fila. A Figura 10 representa um escalonamento de blocos de *threads* diferente para duas GPUs, uma com 2 e outra com 4 multiprocessadores de *streaming*.

Figura 10 – Escalonamento transparente dos blocos de *threads*. Fonte: NVIDIA (2013).



3.3.2 Modelo de memória em CUDA

Desenvolvimento em tecnologias para *hardwares* específicos exigem atenção à hierarquia de memória como o impacto no desempenho da solução (ANGRISH; GARG, 2011). Para obter um bom desempenho em GPU é necessário entender sua hierarquia de memória. O modelo hierárquico de memória dos processadores gráficos utilizados pelo CUDA (Figura 11) é classificado em cinco tipos, cada um com suas características próprias:

1. Registradores: É o tipo de memória mais rápido da GPU, possuem uma grande largura de banda e baixa latência. Cada *thread* tem o seu próprio registrador e só ela pode acessá-lo.

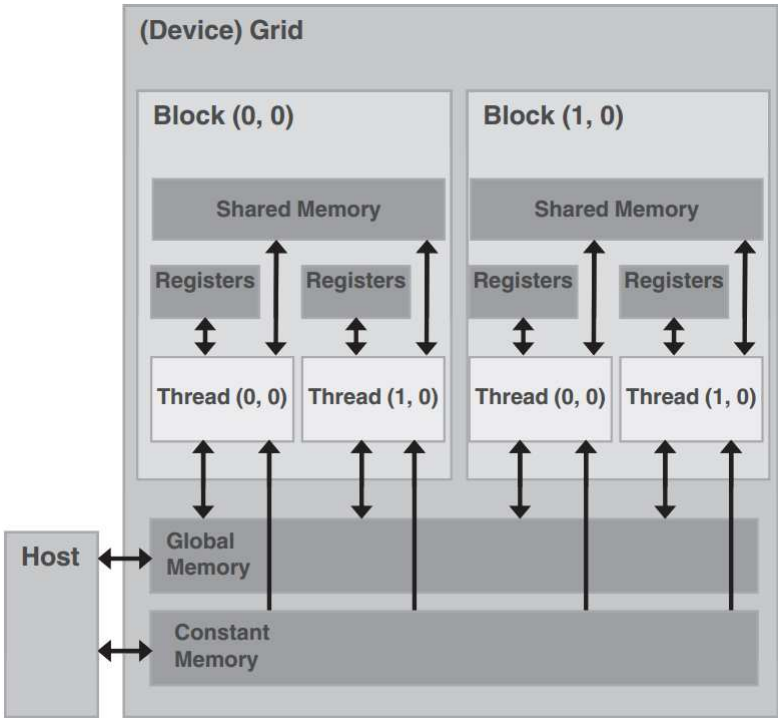
2. Memória local: É utilizada apenas por variáveis automáticas criadas nas funções do *device*.

3. Memória compartilhada: Memória de alta velocidade, otimizada para comunicação rápida entre *threads*. Apenas *threads* do mesmo bloco podem acessá-la.

4. Memória global: Memória que têm alta capacidade de armazenamento, porém tem alto custo de leitura e escrita. Esse tipo de memória pode ser acessado por qualquer *thread*.

5. Memória constante: Memória rápida, apenas de leitura, que pode ser acessada por todas as *threads*. Essa memória tem baixa latência quando todas as *threads* acessam simultaneamente o mesmo local.

Figura 11 – Modelo de memória em CUDA. Fonte: Kirk (2011).



4 Ordenação em GPU

Embora alguns considerem a classificação como um problema resolvido, novos algoritmos são fundamentais para explorar características de arquiteturas atuais. Biologia computacional, motores de busca, renderização em tempo real e sistemas de informações geográficas são áreas atuais onde a ordenação eficiente de grandes quantidades de dados é indispensável (LEISCHNER, 2010).

Como apresentado no Capítulo 3, a GPU é uma ótima plataforma para acelerar a resolução de diversos problemas da computação. Um destes é o problema da ordenação. As próximas seções descrevem as principais metodologias e estratégias para classificação paralela em GPU.

4.1 Metodologias de programação para GPUs

Em Leischner (2010) são apresentados duas principais abordagens:

4.1.1 Projeto de algoritmos SIMD-conscious vs. SIMD-oblivious

É possível ignorar as propriedades SIMD de GPUs e tratar um bloco de *threads* como uma máquina de acesso aleatório paralelo (PRAM). Isso ajuda na criação de algoritmos que são simples e fáceis de portar para outros tipos de arquiteturas paralelas. Estes algoritmos são classificados como SIMD-*oblivious*. No entanto, pode ser vantajoso, fazer uso consciente da arquitetura SIMD. A desvantagem de programação SIMD-*conscious* é que os algoritmos se tornam mais complexos e específicos de hardware. Algoritmos SIMD-*oblivious* são mais comuns, uma vez que eles tendem a ser menos específicos de *hardware* e fáceis de implementar. No entanto, para explorar o paralelismo de GPUs mais eficientemente, o uso de SIMD-*conscious* é essencial.

4.1.2 Projeto de algoritmos multiprocessador-conscious vs. multiprocessador-oblivious

Determinar as dimensões de blocos de *threads* mais apropriadas para um multiprocessador de GPU, pode representar uma melhora significativa de desempenho. Porém isso não é uma tarefa trivial. É difícil determinar os parâmetros ótimos em tempo de execução.

Embora CUDA ofereça um mecanismo extenso de informações sobre o dispositivo em uso, existem algumas limitações. A informação sobre a quantidade de registradores

não é disponibilizada em tempo de execução, sendo necessário consultar o documento de especificação da placa gráfica. O número de registradores disponíveis para multiprocessadores em uma GPU é um parâmetro importante para a configuração de quantos blocos de *threads* podem ser executados ao mesmo tempo.

O uso de multiprocessador-*conscious* leva a uma melhor utilização de hardware. Porém esta abordagem leva a escolhas de *design* específicas de *hardware*. Uma configuração pode ter bom desempenho para algumas GPUs, porém quebrar em outras arquiteturas. Para uma abordagem mais generalizada, multiprocessador-*oblivious* funciona melhor.

4.2 Estratégias básicas para em classificação em GPU

Ainda não há uma classificação formal para as estratégias utilizadas pelos algoritmos de ordenação paralela para GPU. Entretanto notam-se dois modelos de projeto principais que se repetem nos diversos trabalhos disponíveis.

4.2.1 Algoritmos de ordenação baseados em mesclagem

Essa abordagem divide os dados em subsequências menores, ordena esses pedaços de forma independente e por fim, em uma série de etapas mescla as subsquências até se tornarem um único conjunto ordenado. A Figura 12 faz uma ilustração deste modelo.

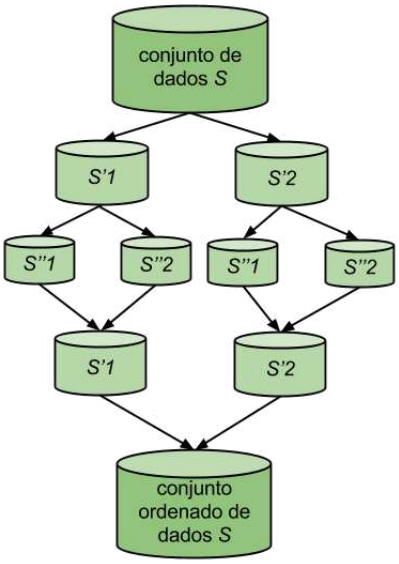


Figura 12 – Modelo de um algoritmo de ordenação baseado em mesclagem

Uma escolha natural para essa abordagem é o mergesort, que possui um excelente balanceamento de carga implícito. Mas diferentes princípios de mesclagem de dados, como bitonic merge, odd-even merge também são utilizados (YANG; DU; ZHANG, 2012).

A fusão de dois conjuntos classificados em um único conjunto ordenado é simples em computação sequencial, mas apresenta desafios quando realizados em paralelo (Green, Oded and McColl, Robert and Bader, 2012). Em um ambiente paralelo uma escolha comum é dividir os dados em subsequências menores de acordo com a quantidade de processadores. Cada processador ordena sua parte através de algum algoritmo sequencial de ordenação. As subsequências vão sendo mescladas de duas em duas por processadores diferentes. Porém, a cada passo de mesclagem, os recursos do ambiente paralelo são mais requisitados individualmente. Pois, em cada passo, a quantidade de subsequências e processadores utilizados para mesclar são reduzidos pela metade e o tamanho das subsequências é dobrado.

Davidson et al. (2012) desenvolveu uma solução eficiente para trabalhar com esses grandes conjuntos gerados das etapas finais do processo de mesclagem. Atualmente, ele é um dos principais trabalhos baseado em ordenação por mesclagem.

4.2.2 Algoritmos de ordenação baseados em distribuição

A primeira fase deste método particiona os dados de entrada em n subconjuntos distintos, cada dado atribuído a um subconjunto s_i deve ser maior do que todos os dados que pertencem a qualquer subconjunto $s_0, \ldots, s_{i-1}$, e menor do que qualquer dado pertencente a um subconjunto $s_{i+1}, \ldots, s_{n-1}, s_n$. Na segunda etapa, os subconjuntos são ordenados independentemente por um algoritmo de classificação e então são unificados reformando um único conjunto. A Figura 13 faz uma ilustração deste modelo.

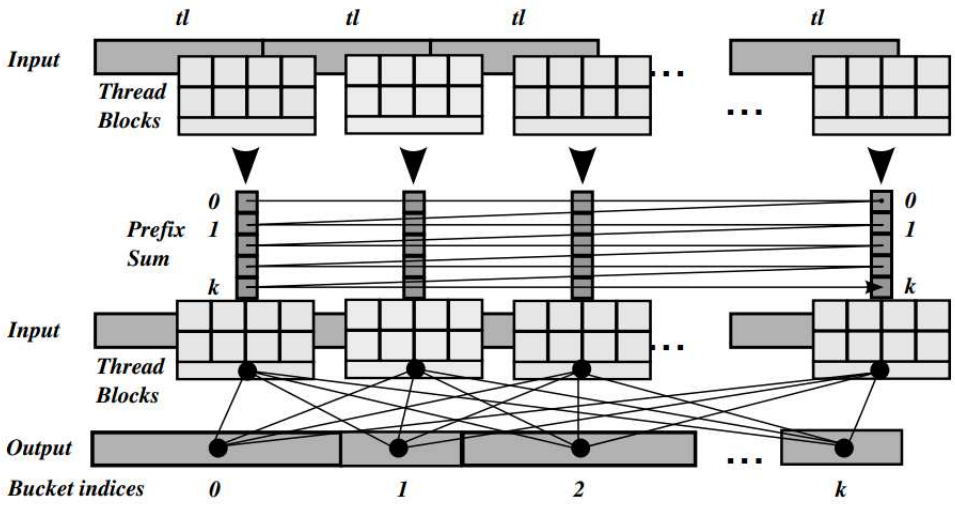


Figura 13 – Modelo de um algoritmo de ordenação baseado em distribuição. Fonte: Leischner, Osipov e Sanders (2009).

Este procedimento é aplicado recursivamente para os subconjuntos, até que eles

tenham um tamanho m. A maioria dos algoritmos de ordenação em GPU mudam para diferentes métodos de classificação quando m se encaixa completamente na memória compartilhada de um SM. Normalmente algoritmos mais simples, como *insertion sort*, são uma escolha popular.

Os subconjuntos são as estruturas intermediárias nas quais os dados são distribuídos. Eles são definidos de acordo com as características dos dados (*radix sort*) ou através de amostras do conjunto total de dados (*sample sort*).

Dos principais métodos de ordenação para GPU, dois são classificadores baseados em divisão, o Satish, Harris e Garland (2009) apresenta um *radix sort* que tem um desempenho excelente para dados inteiros de 32 *bits*, enquanto o Leischner, Osipov e Sanders (2009) desenvolveu um *sample sort* eficiente para dados inteiros acima de 32 *bits*.

Parte II

Projeto

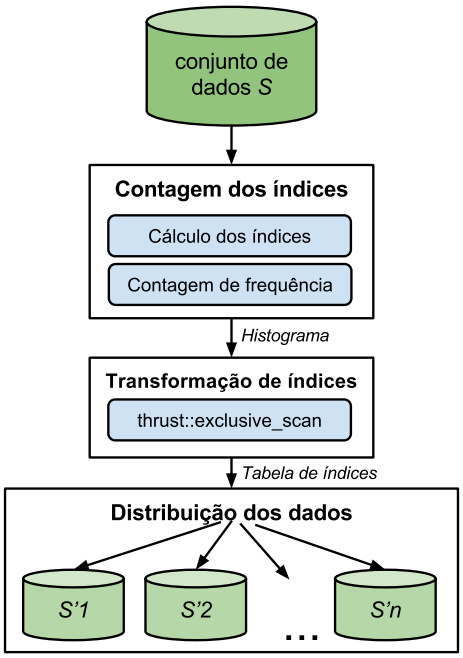
5 GPU MSD Radix String Sort

Como visto na Seção 2.3.4, algoritmos de ordenação de *strings* baseados em comparação tem limite inferior de $O(d+n \log n)$, onde d representa o número mínimo de caracteres necessários que precisam ser analisados para que o critério de ordem seja estabelecido. Quanto mais dados compartilharem e mais longo forem os prefixos compartilhados por eles, maior o valor de d e mais significância ele tem na função de complexidade. Para evitar repetições de comparações desnecessárias, alguns algoritmos adotam abordagens específicas para se aproximar de uma complexidade ótima.

Embora não seja a finalidade principal, a ordenação por distribuição evita a repetição dessas comparações desnecessárias. *Radix sorts* são teoricamente atraentes, pois os primeiros caracteres em cada *string* que são usados para alocar a *string* em um subconjunto, são inspecionados uma única vez (SINHA; ZOBEL, 2003).

A fim de reduzir o custo de comparações repetidas desnecessárias e otimizar o desempenho, este projeto adotou o modelo do *MSD Radixsort*, distribuindo as suas principais etapas para serem processadas por múltiplas *threads* em paralelo. A Figura 14 ilustra a visão geral dessa fase do projeto.

Figura 14 – Visão ampla do projeto do algoritmo. Fonte: Elaborada pelo Autor.



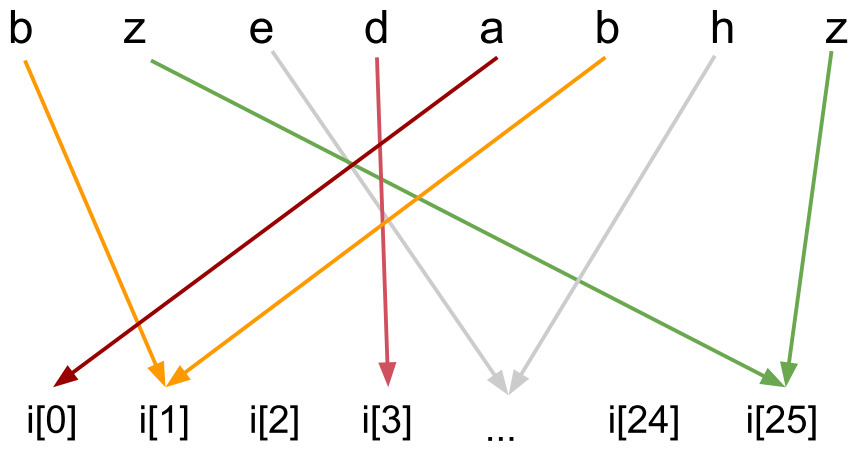
5.1 Contagem dos índices

O primeiro passo deste algoritmo é a contagem dos índices das *strings*. Inicialmente é necessário transformar a *string* em um índice. Depois, acrescenta-se em um ao contador global da sua respectiva posição. Logo, o resultado final desta etapa é um histograma que representa a quantidade de *string* para cada índice. Os passos de cálculo de índice e contagem de frequência são descritos em seguida.

5.1.1 Cálculo de índice

O cálculo de índice, de uma maneira simplificada, é uma etapa de transformação de uma *string* em um índice. Na Figura 15 podemos ver um exemplo simples desta transformação. Nesta ilustração, o alfabeto considerado das *strings* é o conjunto de letras utilizados pelo idioma português. Cada palavra iniciada por uma letra é representada por um índice. Logo, para este exemplo, existem 26 índices.

Figura 15 – Exemplo de uma transformação de strings em índices. Fonte: Elaborada pelo Autor.



Algoritmos de distribuição possuem diferentes técnicas para transformar *strings* em índices. O *Radix sort*, utiliza a própria característica dos dados para definir quais são os seus índices de subconjunto. Por exemplo, para uma cadeia de caracteres de DNA, os valores de subconjuntos possíveis seriam; $\{a, c, g, t\}$. O índice do subconjunto é correspondente à posição dele em uma lista ordenada destes valores, começando de zero. Logo, os respectivos índices para os valores de um carácter de DNA, são; $\{a = 0, c = 1, g = 2, t = 3\}$.

O Algoritmo 5.1 mostra a função utilizada que calcula o índice de um caractere para um determinado alfabeto. Este método espera como parâmetro, um conjunto ordenado de valores de um alfabeto (*dictionary*), a quantidade de valores desse alfabeto e um caractere. O valor retornado corresponde ao índice desse caractere no alfabeto.

Algoritmo 5.1: Função que calcula o índice de um caractere para um alfabeto qualquer em CUDA

```
__device__ int searchIndexDictionary(char *dictionary,
                            int size_dictionary, char ch) {
  int i;
  for(i=0; i<size_dictionary; i++)        {
    if(dictionary[i] == ch) {
      return i;
    }
  }
  return -1;
}
```

Afim de evitar repetidos custos de alocação e transferência de dados e otimizar o desempenho, às vezes é necessário calcular o índice para um bloco de caracteres. Para que o Algoritmo 5.1 funcione para estes casos, seria necessário que o dicionário contemplasse a junção desses blocos de caracteres. Por exemplo, para o cálculo de índice de dois caracteres de uma cadeia de DNA, o dicionário teria que ser: $\{aa, ac, ag, at, \ldots, ta, tc, tg, tt\}$.

Uma solução mais elegante seria como no Algoritmo 5.2, que calcula o índice do bloco através da projeção de cada carácter do bloco em um cálculo de potência.

Algoritmo 5.2: Função que calcula o índice de um bloco de caractere em CUDA

```
__device__ int searchIndex(STRING word, int depth,
             char *dictionary, int size_dictionary) {
  int i, index, col;

  index = 0;
  for(i=0; i<depth && word[i]!='\0'; i++)        {
   col = searchIndexDictionary(dictionary, size_dictionary, word[i]);
   index += col * pow((float)size_dictionary, (float) (depth-(i+1)));
  }
  return index;
}
```

Como a manipulação de *strings* tem um custo elevado, é necessário trabalhar com elas ao invés de *substrings*. Por isso, o Algoritmo 5.2, espera além da *string*, a profundidade (*depth*) dos elementos que serão inspecionados, como parâmetro.

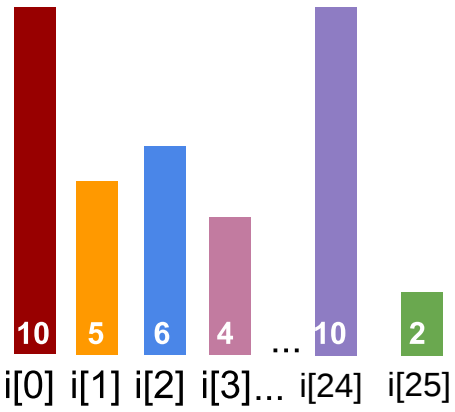
5.1.2 Contagem de frequência

Após o passo de transformação da *string* em índice, realizado na etapa anterior, é feito a contagem de frequência deste índice. A contagem de frequência dos índices é garantida através de uma estrutura global que armazena a quantidade de *strings* ocorridas para cada índice. Este passo é mais simples e requer apenas uma operação que soma o valor de 1 (um) no contador correspondente à posição de índice de uma *string*. Essa soma é garantida através de uma função atômica, que evita o problema de concorrência das múltiplas *threads* que acessem esta mesma posição de memória. A função atômica apresentada no Algoritmo 5.3 garante que todas as *threads* mantenham-se com os contadores de índices sempre atualizados. Após todas as *strings* serem transformadas em índices e feito a contagem desses índices, um histograma do conjunto analisado é obtido e a primeira parte do algoritmo é finalizada. A Figura 16 ilustra um histograma com a contagem de frequência de um conjunto qualquer. Neste histograma, existem 8 elementos pertencentes ao primeiro índice, 5, 6, e 4 elementos pertencentes aos próximos três índices respectivamente, e por fim, o último índice possui 2 elementos.

Algoritmo 5.3: Trecho de código da contagem de frequência

```
1 //bckts[index]++
2 atomicAdd(bckts + index, 1);
```

Figura 16 – Exemplo de um histograma. Fonte: Elaborada pelo Autor.

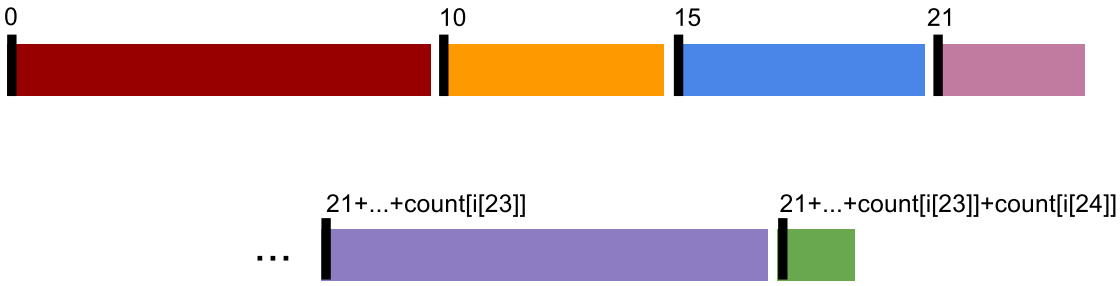


5.2 Transformação das contagens de frequência em índice

Para que os dados possam ser redistribuídos e assim estarem ordenados, é necessário saber qual a sua nova posição no conjunto final. Logo, após a conclusão deste passo é obtido uma sequência com as posições iniciais de cada índice no conjunto final.

Após os cálculos dos índices de todas as *strings*, o histograma com a contagem de frequência dos índices está pronto. É a partir dele que a transformação dos índices será feita. Nesta etapa é feito uma horizontalização do histograma, onde a quantidade de elementos de cada índice representa a quantidade de memória que deve ser reservada para os índices de cada *string*. Um exemplo de horizontalização do histograma da Figura 16 pode ser visto na Figura 17.

Figura 17 – Exemplo de horizontalização de um histograma. Fonte: Elaborada pelo Autor.



Em geral, para obter o índice inicial de uma *string*, basta somar as contagens de frequência das *strings* antecessoras a ela. O índice inicial de uma *string* b_{n+1} é igual à soma de todas as contagens das *strings* antecessoras a ela, $b_0, b_1, \ldots, b_n$. Esta é uma tarefa frequente na computação, e é chamada de soma de prefixos. Neste projeto é utilizada a função de soma de prefixo fornecido pela biblioteca *thrust* (THRUST, 2012) do CUDA, por já apresentar um resultado satisfatório para GPUs. Um exemplo de chamada dessa função pode ser vista no Algoritmo 5.4.

Algoritmo 5.4: Chamada à função de soma de prefixo da biblioteca thrust do CUDA

```
1 #include <thrust/scan.h>
2
3 int data[6] = {1, 0, 2, 2, 1, 3};
4
5 thrust::exclusive_scan(data, data + 6, data); // in-place scan
6
7 // data is now {0, 1, 1, 3, 5, 6}
```

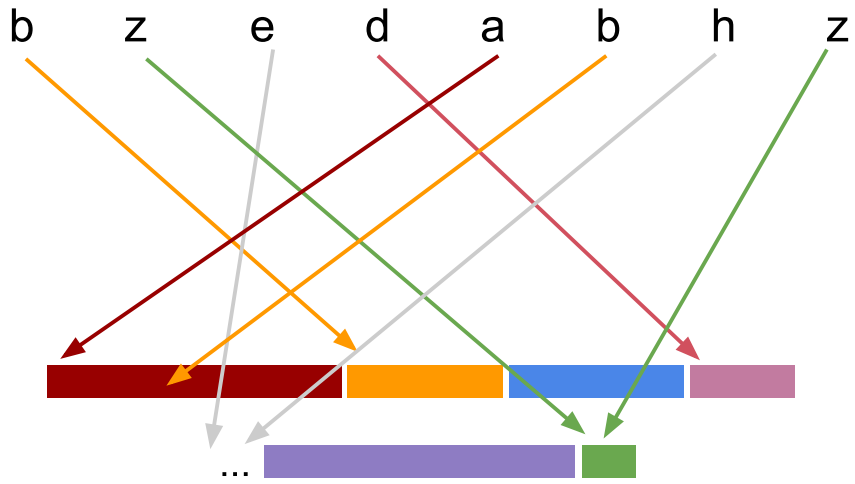
5.3 Distribuição dos dados

A etapa de redistribuição dos dados, é a última etapa do algoritmo que produz um resultado ordenado do conjunto de *strings*.

Com a tabela de índices iniciais gerada pela etapa anterior, a redistribuição é feita, movendo as *strings* para sua posição em uma matriz auxiliar, como demonstra a Figura

18. Cada *string* através do seu índice acessa a tabela para saber qual a sua posição no conjunto final. Após consultar a posição de uma *string* no conjunto final, automaticamente é incrementado em um, através de uma função atômica, o valor de índice inicial da tabela para essa entrada. Assim, garante que a próxima *string* de mesmo índice acesse a sua posição no conjunto final corretamente.

Figura 18 – Exemplo de distribuição dos dados. Fonte: Elaborada pelo Autor.



Por fim, o resultado ordenado é copiado de volta do *device* para a matriz original que está armazenada no *host*.

Parte III

Resultados experimentais

6 Experimentos e Resultados

6.1 Descrição dos experimentos

Os resultados deste trabalho foram obtidos através de três experimentações distintas.

A primeira avalia o desempenho do algoritmo deste trabalho em diferentes configurações de parâmetros; quantidade de dados analisados por *thread* e estruturação dos blocos de *threads* na GPU.

Na segunda experimentação, é comparado o desempenho do algoritmo desenvolvido neste trabalho com o estado da arte de ordenação de *strings*, segundo o tempo de execução de cada um. Como não foi encontrada nenhuma biblioteca ou código disponível para ordenação de *strings* em GPU, o algoritmo desenvolvido neste trabalho será comparado com os algoritmos da biblioteca de Bingmann e Sanders (2013), que ordena *strings* para arquiteturas *multi-core*.

A última experimentação também avalia o desempenho do algoritmo proposto, porém em um teste com conjuntos de dados de tamanho fixo, variando a quantidade de prefixos compartilhados pelas *strings*.

6.1.1 Algoritmos comparados

Como falado anteriormente, o algoritmo deste trabalho foi comparado com algoritmos da biblioteca de Bingmann e Sanders (2013). Esta biblioteca contém algoritmos clássicos e atuais de ordenação de *strings* sequenciais e algoritmos paralelizados para sistemas *multi-core*, que foram utilizados e experimentados no trabalho de Bingmann e Sanders (2013).

Para os testes, foram escolhidos dois algoritmos desta biblioteca:

1. Parallel sample sortBTCU2: é o algoritmo de melhor resultado para diversos conjuntos de dados descrito em Bingmann e Sanders (2013);
2. *Parallel radix sort 16bit*: uma abordagem similar ao algoritmo deste trabalho, porém para *multi-core*.

6.1.2 Conjunto de dados

Para avaliar os algoritmos em conjuntos de ambientes reais e que tivessem alta taxa de prefixos comuns, foram utilizados conjuntos de dados com alfabetos pequenos. Os

dados foram gerados para bases de cadeias genômicas.

Os dados genômicos são um conjunto de sequências de nucleotídeos e têm um alfabeto de quatro caracteres $\{a, c, g, t\}$. Neste trabalho, eles são analisados em cadeias mais curtas, cadeias de 9 caracteres de comprimento. Essas cadeias de 9 caracteres de comprimento são comumente usadas por utilitários de pesquisa genômica(HEINZ et al., 2002).

Grande parte dos testes foram feitos para conjuntos de dados *strings* de tamanhos, aproximadamente: $1,5 \times 10^6$, $7,6 \times 10^6$, 38×10^6 e 153×10^6. Para avaliar o desempenho do algoritmo em ambientes com alto índice de repetição de prefixos, o tamanho do problema foi fixado. Três conjuntos de 38×10^6 *strings* foram criados, com variação de 1, 4 e 8 de comprimento mínimo do prefixo compartilhado por todas elas.

As Tabelas 1 e 2 detalham as características destes conjuntos.

Tabela 1 – Conjuntos de dados com distribuição uniforme

	Conjunto 1	Conjunto 2	Conjunto 3	Conjunto 4
Tamanho do conjunto (MB)	15,4	76,8	384,0	1.536,0
Qtd. de strings	1.536.000	7.680.000	38.400.000	15.360.000
Qtd. de strings distintas	262.144	262.144	262.144	262.144

Tabela 2 – Conjuntos de dados com variação de strings distintas

	Conjunto 5	Conjunto 6	Conjunto 7
Tamanho do conjunto (MB)	384,0	384,0	384,0
Qtd. de strings	38.400.000	38.400.000	38.400.000
Qtd. de strings distintas	65.536	1.024	4

6.1.3 Medição do tempo

Para computar o tempo do código executado na CPU, foi utilizada a função *clock* da biblioteca *timer.h* da linguagem *C*. A computação do tempo das funções *kernel* do CUDA teve de ser calculada através da criação de eventos, como mostra o Algoritmo 6.2.

Algoritmo 6.1: Exemplo de uso da função clock da biblioteca time.h

```
#include <stdio.h>
#include <time.h>
int main ()
{
  clock_t t;
  t = clock();
  //...
```

```
8   t = clock() - t;
9   printf("%f seconds.\n",t);
10  return 0;
11 }
```

Algoritmo 6.2: Trecho de código para computar o tempo de uma função device do CUDA

```
1  float cputime;
2  cudaEvent_t start, stop;
3
4  cudaEventCreate(&start);
5  cudaEventCreate(&stop);
6
7  cudaEventRecord(start,0);
8  //call function device;
9  cudaEventRecord(stop, 0);
10 cudaEventSynchronize(stop);
11
12 cudaEventElapsedTime(&cputime, start, stop);
13
14 cudaEventDestroy(start);
15 cudaEventDestroy(stop);
```

6.1.4 Repetição dos testes

Os testes foram repetidos três vezes para observar se houve alguma variação de tempo durante a execução dos algoritmos. Os melhores resultados foram preservados.

6.1.5 Ambiente de experimentação

Os testes dos algoritmos deste trabalho foram realizados em um ambiente Ubuntu x86_64 GNU/Linux, com 24 processadores Intel(R) Xeon(R) CPU E5-2620 0 @ 2.00GHz e placa gráfica NVIDIA Tesla C2075 com um total de 448 cores, 14 multiprocessadores de *threads* com 32 processadores escalares em cada multiprocessador e com uma largura de banda da memória de 144GB/s. Este equipamento se encontra na Universidad de Murcia, Espanha.

6.2 Validação

A saída de cada algoritmo de ordenação foi verificada em duas etapas, explicadas a seguir.

6.2.1 Validação da integridade do conjunto

A primeira etapa verificou a integridade dos conjuntos, verificando se a lista de saída era uma permutação do conjunto de entrada através da comparação dos histogramas dos mesmos.

6.2.2 Validação da ordenação

Na segunda etapa foi verificado se as strings estavam em ordem lexicográfica, como pode ser visto no Algoritmo 6.3.

Algoritmo 6.3: Verificação de ordem lexicográfica de uma lista de strings

```
1  int check_sorted(STRING strings, int max_length, int size) {
2    int i;
3    //check order lexicographic
4    for(i=0; i<size-1; i++) {
5      if(!strLess(strings + i*max_length, strings + (i+1)*max_length)) {
6        return 0;
7      }
8    }
9    return 1;
10 }
```

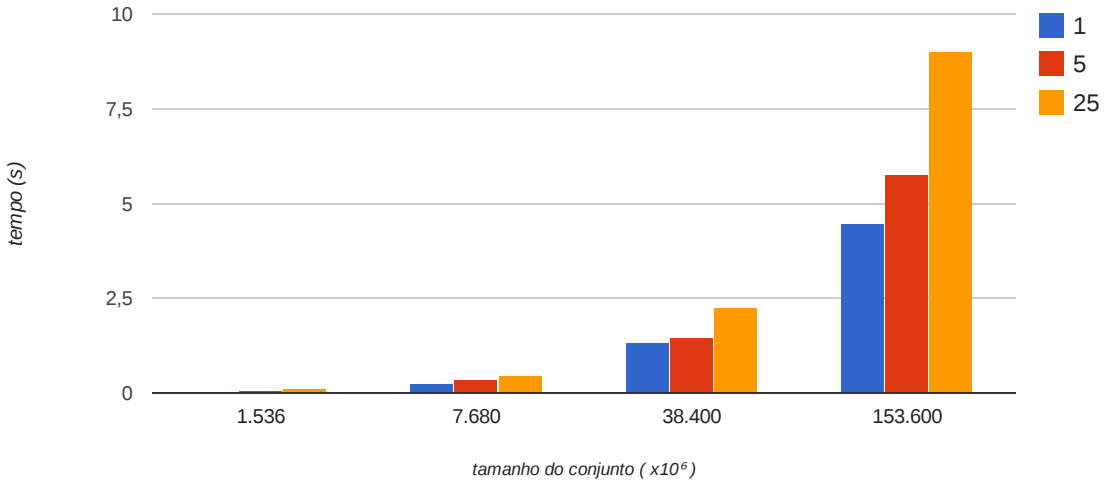
6.3 Resultados

As Tabelas 3 e 4 apresentam os resultados do algoritmo deste trabalho para diferentes configurações de quantidades de *strings* analisadas por *threads* e quantidade de *threads* por blocos, respectivamente. Os resultados da Tabela 3 demonstram que o algoritmo obtêm melhores resultados, para todos os conjuntos testados, quando cada *thread* analisa apenas uma *string* por vez. Na Figura 19 percebe-se que quanto menor o número de *strings* analisadas por *thread*, melhor é o desempenho do algoritmo. Essa diferença se dá por causa da forma de acesso à memória. Quanto mais elementos cada *thread* analisa, mais espaçado é o acesso das *threads* à memória, e a GPU tem melhor eficiência quando à memória é acessada de forma contígua (LEISCHNER; OSIPOV; SANDERS, 2009).

Tabela 3 – Desempenho do GPU msd radix string sort com variação da quantidade de dados analisados por threads para diferentes conjuntos (segundos).

Quantidade de dados	Conjunto 1	Conjunto 2	Conjunto 3	Conjunto 4
1	0,05	0,26	1,33	4,47
5	0,07	0,36	1,47	5,79
25	0,12	0,47	2,27	9,01

Figura 19 – Desempenho do GPU msd radix string sort com variação da quantidade de dados analisados por threads.

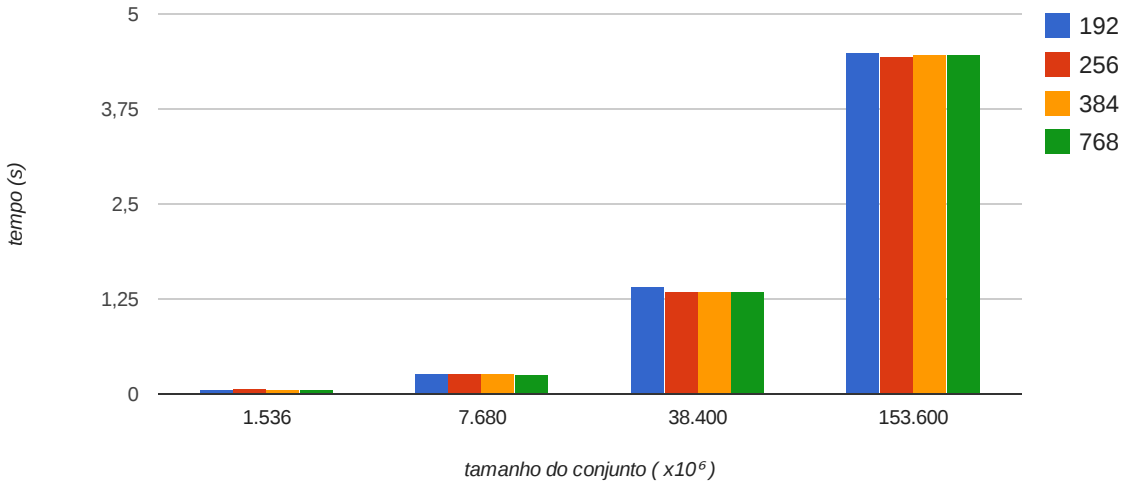


A GPU utilizada neste trabalho, suporta até 1.536 *threads* por multiprocessador, e apenas 1.024 *threads* por cada bloco executado no multiprocessador. Os resultados da Tabela 4, ilustrados na Figura 20, demonstram que o algoritmo não teve significativas variações para diferentes quantidades de *threads* analisadas por cada bloco. O que indica que, para este algoritmo, o mais relevante não é a quantidade de *threads* executadas por bloco, mas sim, a organização dos blocos para utilizar o máximo de threads possíveis do multiprocessador.

Tabela 4 – Desempenho do GPU msd radix string sort para diferentes números de threads por blocos (segundos).

número de threads	Conjunto 1	Conjunto 2	Conjunto 3	Conjunto 4
192	0,06	0,26	1,40	4,49
256	0,07	0,26	1,33	4,44
384	0,05	0,26	1,33	4,47
768	0,06	0,24	1,33	4,47

Figura 20 – Desempenho do GPU msd radix string sort com variação de threads por blocos.

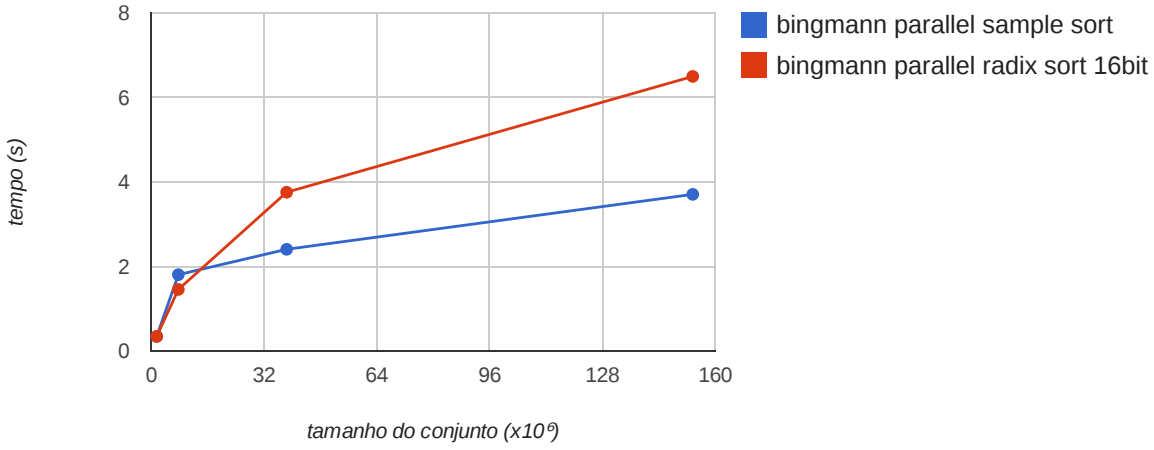


O algoritmo obteve desempenho superior aos algoritmos comparados para todos os conjuntos testados. A aceleração do algoritmo deste trabalho em relação aos algoritmos da biblioteca de Bingmann e Sanders (2013) para os Conjuntos 1, 2, 3 e 4, é apresentada na Tabela 5. Pelos resultados apresentados nesta tabela, nota-se que o índice de aceleração cresce com o tamanho do problema (ver Figura 21). A maior aceleração de desempenho para o *bingmann parallel sample sort* é de quase 4× (vezes), enquanto que para o *bingmann parallel radix sort 16bit* chega a ser mais de 6× (vezes) mais rápido. Como esperado pelos resultados apresentados em Bingmann e Sanders (2013), a aceleração em relação ao *bingmann parallel sample sort* foi menor do que ao do *bingmann parallel radix sort 16bit.*

Tabela 5 – Aceleração do GPU msd radix string sort em comparação com os outros algoritmos.

	Conjunto 1	Conjunto 2	Conjunto 3	Conjunto 4
bingmann parallel sample sort	0,34	1,80	2,40	3,70
bingmann parallel radix sort 16bit	0,34	1,45	3,75	6,49

Figura 21 – Aceleração do GPU msd radix string sort em comparação com os outros algoritmos.



A Tabela 6 detalha o desempenho de cada etapa do algoritmo deste trabalho para os diferentes conjuntos de dados. Os resultados mostram que a etapa mais custosa para os conjuntos 1 e 2 é o passo de seleção da GPU. Etapa em que ocorre a primeira comunicação entre a *host* e o *device*, onde é definida em qual GPU o algoritmo será executado. Por ter um custo fixo, enquanto as outras etapas aumentam o seu custo conforme o tamanho do problema aumenta, esta fase de seleção da GPU passa a ser menos custosa para os conjuntos 3 e 4. Porém, seu custo ainda é relativamente significante para o tempo total do algoritmo. Outra etapa que tem um custo significativo para todos os conjuntos testados, é a fase de transferência do conjunto de dados, onde todos os dados são transferidos da memória da CPU e alocados na memória da GPU. Vale ressaltar que para todos os conjuntos, o passo de transformação de índices manteve-se com o tempo de execução quase fixo e foi a etapa menos custosa para todos os conjuntos testados. As Figuras 22, 23, 24 e 25 ilustram a proporção dos custos de cada etapa para os diferentes conjuntos.

Tabela 6 – Custo das etapas do GPU msd radix string sort para diferentes conjuntos (segundos).

	Conjunto 1	Conjunto 2	Conjunto 3	Conjunto 4
Seleção da GPU	0,510	0,520	0,560	0,520
Transferência dos dados do conjunto	0,060	0,090	0,190	0,550
Custos extras	0,000	0,050	0,320	0,510
Contagem de Índices	0,007	0,031	0,146	0,577
Transformação de índices	0,009	0,011	0,011	0,009
Distribuição dos dados	0,034	0,171	0,856	3,373

Figura 22 – Custo das etapas do GPU msd radix string sort para o Conjunto 1.

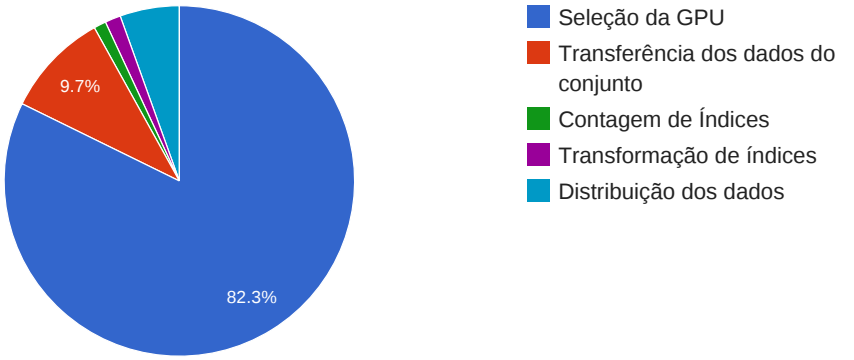


Figura 23 – Custo das etapas do GPU msd radix string sort para o Conjunto 2.

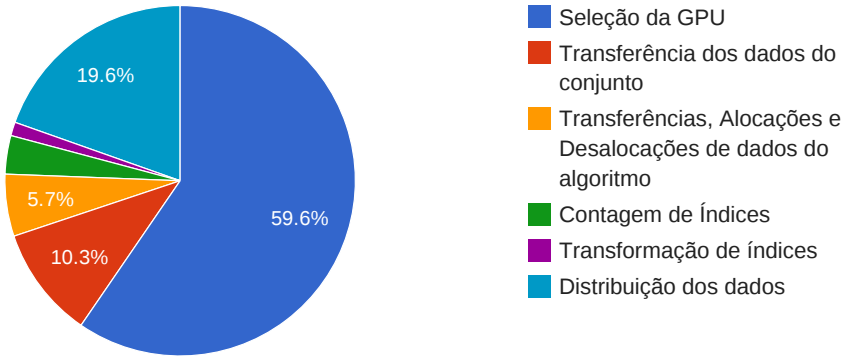


Figura 24 – Custo das etapas do GPU msd radix string sort para o Conjunto 3.

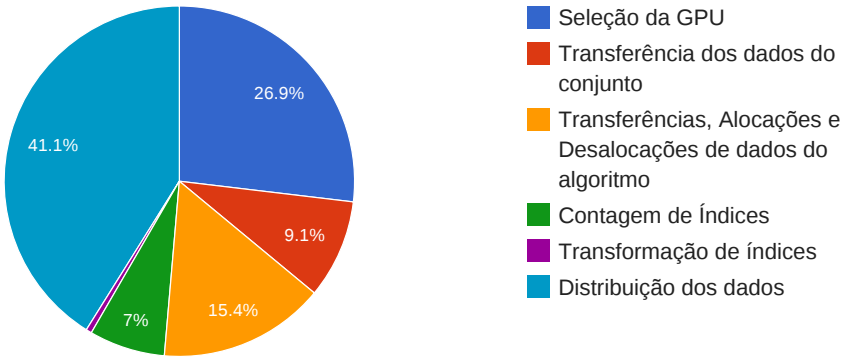
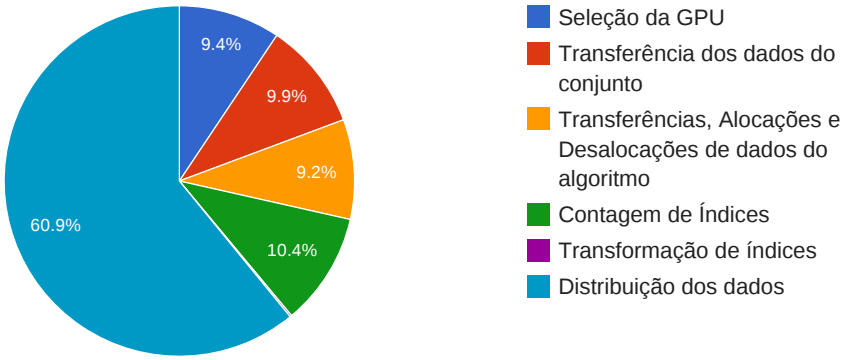


Figura 25 – Custo das etapas do GPU msd radix string sort para o Conjunto 4.



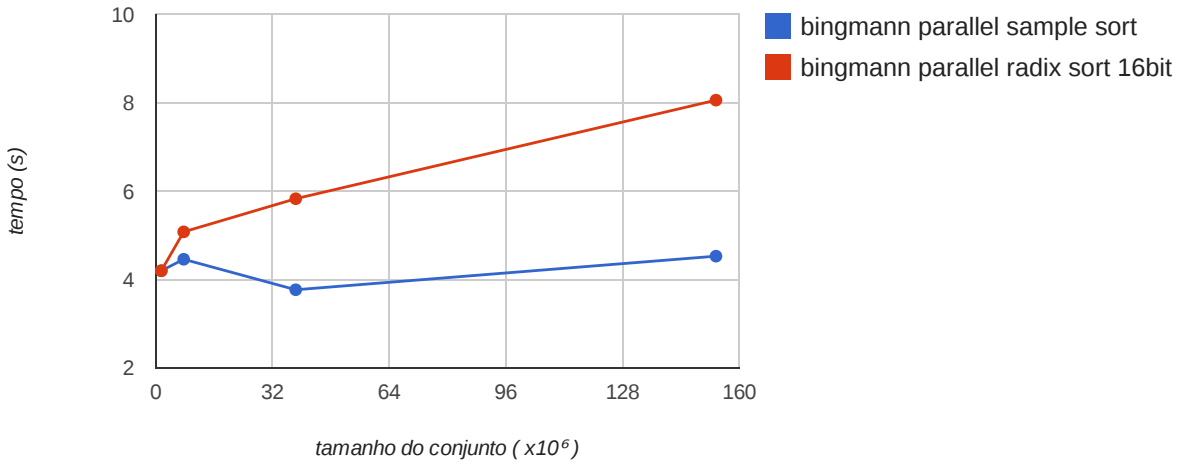
Como visto anteriormente, duas das etapas mais custosas do algoritmo deste trabalho são a de seleção da GPU e de transferência dos dados do conjunto. Estas são etapas do algoritmo que preparam o ambiente para que a ordenação possa ser feita na GPU. Alguns trabalhos importantes (LEISCHNER; OSIPOV; SANDERS, 2009; SATISH; HARRIS; GARLAND, 2009; YANG; DU; ZHANG, 2012; DAVIDSON et al., 2012), não contabilizaram os custos iniciais e de transferências dos dados para a GPU, pois eles consideram que a ordenação é, em grande parte, uma etapa de um algoritmo muito mais elaborado, e por isso, estes custos iniciais são contabilizados por este algoritmo mais elaborado, e não especificamente pela ordenação.

Desconsiderando os tempos dos passos iniciais, a Tabela 7 mostra como ficaram os resultados da aceleração do algoritmo. O algoritmo passa a ter uma aceleração superior a 4× (vezes) para o *bingmann parallel sample sort* em quase todos os conjuntos. Para o *bingmann parallel radix sort 16bit*, a aceleração é sempre superior a 4× (vezes) e chega a 8× (vezes) para o Conjunto 4 (ver Figura 26). É importante ressaltar que os resultados apresentados nas Tabelas 3, 4, 6 e 8, por representarem dados de comparações feitas apenas com o desempenho do próprio algoritmo, os custos das etapas iniciais de comunicação com a GPU não foram considerados.

Tabela 7 – Aceleração do GPU msd radix string sort em comparação com os outros algoritmos sem os custos de comunicações iniciais.

	Conjunto 1	Conjunto 2	Conjunto 3	Conjunto 4
bingmann parallel sample sort	4,20	4,46	3,77	4,53
bingmann parallel radix sort 16bit	4,20	5,08	5,83	8,06

Figura 26 – Aceleração do GPU msd radix string sort em comparação com os outros algoritmos sem os custos de comunicações iniciais.



A Tabela 8 lista os resultados da aceleração do algoritmo deste trabalho para os conjuntos com variação das quantidades mínimas de prefixos compartilhados pelas *strings*. Foi considerado como tempo base, o desempenho do algoritmo deste trabalho para o Conjunto 3, que tem o mesmo tamanho do problema que os Conjuntos 5, 6 e 7, porém é o ambiente de menor compartilhamento de prefixos. Como esperado, o algoritmo apresenta um bom desempenho para ambientes com maior prefixos compartilhados. Porém, a aceleração do desempenho, em um certo momento, chega a ser mais de 2× (vezes) mais rápida que o desempenho do algoritmo para o Conjunto 3 (ver Figura 27). Era esperado que o algoritmo mantivesse desempenho constante ou com pequenas melhorias para tais conjuntos. Para entender como isso ocorreu, foi feita uma análise dos custos das principais etapas do algoritmo para este ambiente. A Tabela 9 e a Figura 28 ilustram as proporções destes custos de acordo com os conjuntos. Ao observar a Figura 28, nota-se que a etapa de distribuição dos dados, uma das mais custosas para os demais conjuntos, reduz o tempo significativamente conforme o conjunto contém mais *strings* semelhantes. Mais uma vez, essa diferença se dá por causa da forma de acesso à memória. Como dito anteriormente, acessos contíguos à memória da GPU são mais eficientes que os acessos não contíguos. A etapa de distribuição é a etapa que além de fazer acessos de leitura à memória também faz acessos de escrita, sendo uma das etapas mais custosa do algoritmo. Como quanto maior foi a repetição de *strings* de um conjunto, mais próximas essas *strings* estavam organizadas na memória, o custo da etapa de distribuição dos dados foi inversamente proporcional à quantidade de *strings* repetidas.

Tabela 8 – Aceleração de desempenho do GPU msd radix string sort para diferentes conjuntos.

Conjunto 3	Conjunto 5	Conjunto 6	Conjunto 7
1,00	1,02	1,79	2,51

Figura 27 – Aceleração de desempenho do GPU msd radix string sort para os Conjuntos 5, 6 e 7.

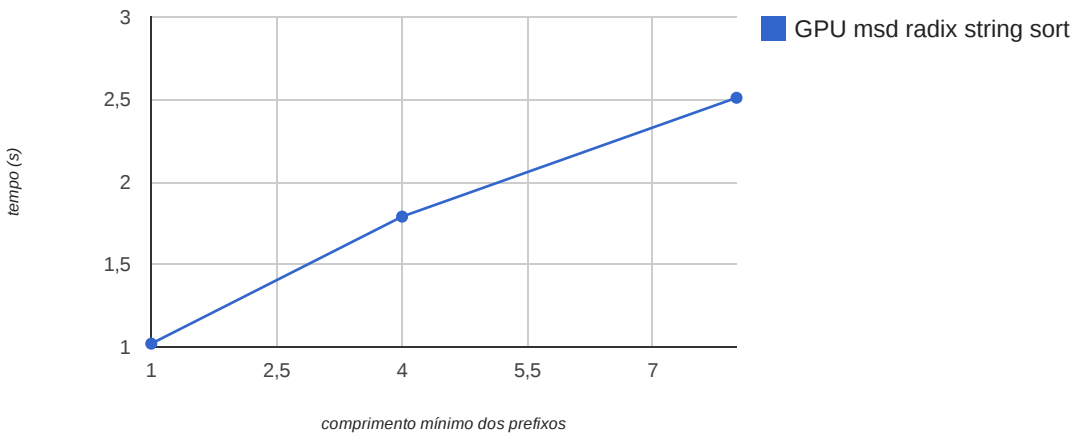
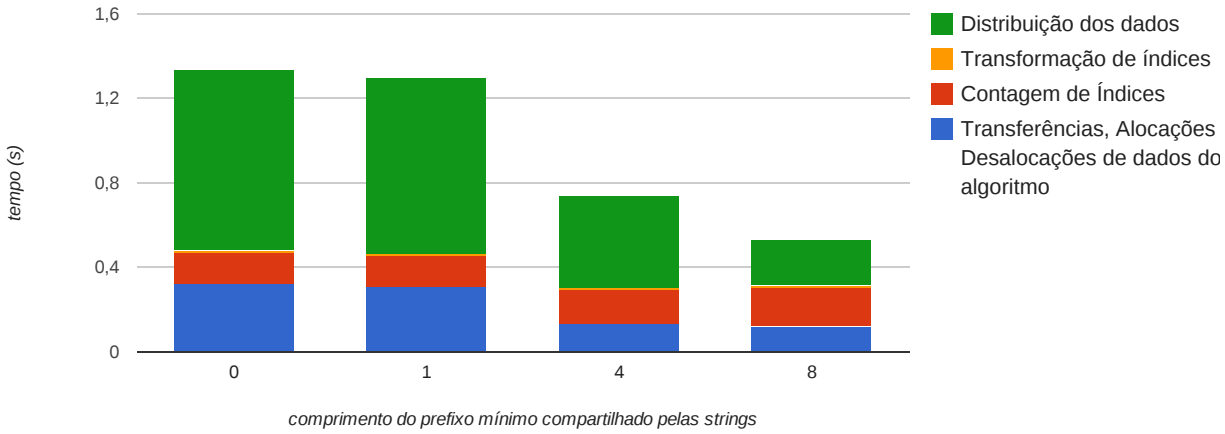


Tabela 9 – Custo das etapas do GPU msd radix string sort para conjuntos de diferentes quantidades mínimas de prefixos compartilhados (segundos).

	Conjunto 3	Conjunto 5	Conjunto 6	Conjunto 7
Custos extras	0,320	0,310	0,130	0,120
Contagem de Índices	0,146	0,143	0,165	0,180
Transformação de índices	0,011	0,011	0,009	0,011
Distribuição dos dados	0,856	0,832	0,434	0,220

Figura 28 – Custo das etapas do GPU msd radix string sort para os Conjunto 3, 5, 6 e 7.



Conclusão

A classificação é um problema muito estudado na computação, porém existem poucos algoritmos que resolvam o problema de ordenação de *strings* para GPU. Davidson et al. (2012) desenvolveu um algoritmo de ordenação de *strings* eficiente, porém com pior desempenho para ambientes com altos índices de prefixos compartilhados. Os resultados experimentais do presente trabalho indicam que, para ambientes de alfabetos pequenos, como cadeias genômicas, e ambientes com alta repetição de prefixos, o algoritmo proposto é eficiente, chegando a ter mais do dobro de eficiência para o conjunto testado com maior ocorrência de prefixos compartilhados por *strings*. Além disso, o algoritmo chegou a ser de quatro a oito vezes mais rápido que os algoritmos *bingmann parallel sample sort* e *bingmann parallel radix sort 16bit* de Bingmann e Sanders (2013), repectivamente.

A ordenação de dados *strings* em GPU ainda representa um amplo campo de pesquisa para se aprimorar, possibilitando estudos de novos caminhos que otimizem o desempenho dos algoritmos para diferentes conjuntos de dados. Como perspectiva para continuação deste trabalho pode-se destacar: testes mais amplos, que abordem alfabetos de tamanhos maiores e chaves de comprimento variável; implementação de uma outra abordagem de ordenação baseada em distribuição eficiente, como o *parallel sample sort* de Bingmann e Sanders (2013), para GPU, e compará-la com a abordagem *MSD radix sort*, utilizada neste trabalho.

Referências

ANGRISH, R.; GARG, D. Efficient String Sorting Algorithms: Cache-aware and Cache-Oblivious. *International Journal of Soft Computing and ...*, n. 2, p. 12–16, 2011. Disponível em: <http://www.gdeepak.com/pubs/EfficientStringSortingAlgorithmsCache-awareandCache-Oblivious.pdf>. Citado 2 vezes nas páginas 20 e 31.

BEETS, K. *GPU compute – what is it and why do we care?* 2013. Disponível em: <http://withimagination.imgtec.com/index.php/powervr/gpu-compute-what-is-it-and-why-do-we-care>. Citado na página 27.

BENTLEY, J.; SEDGEWICK, R. Fast algorithms for sorting and searching strings. In: *Proceedings of the eighth annual ACM-SIAM symposium on Discrete algorithms*. Philadelphia, PA, USA: Society for Industrial and Applied Mathematics, 1997. p. 360–369. Disponível em: <http://dl.acm.org/citation.cfm?id=314321>. Nenhuma citação no texto.

BINGMANN, T.; SANDERS, P. Parallel String Sample Sort. *CoRR*, abs/1305.1, 2013. Disponível em: <http://arxiv.org/abs/1305.1157>. Citado 5 vezes nas páginas 12, 21, 45, 50 e 56.

CHEN, S. et al. A fast and flexible sorting algorithm with CUDA. In: *Proceedings of the 9th International Conference on Algorithms and Architectures for Parallel Processing*. Berlin, Heidelberg: Springer-Verlag, 2009. p. 281–290. Disponível em: <http://dl.acm.org/citation.cfm?id=1615035>. Nenhuma citação no texto.

DANTAS, A. C. d. C. Avaliação de algoritmos de ordenação em sistemas paralelos. 1997. Disponível em: <http://libdigi.unicamp.br/document/?code=vtls000129219>. Citado 2 vezes nas páginas 12 e 15.

DAVIDSON, A. et al. Efficient Parallel Merge Sort for Fixed and Variable Length Keys. In: *2012 Innovative Parallel Computing (InPar)*. San Jose, CA: [s.n.], 2012. p. 1–9. ISBN 978-1-4673-2633-9. Disponível em: <http://www.idav.ucdavis.edu/func/return_pdf?pub_id=1085>. Citado 5 vezes nas páginas 12, 16, 35, 53 e 56.

FILHO, C. F. *História da computação: O Caminho do Pensamento e da Tecnologia*. EDIPUCRS„ 2007. 205 p. ISBN 9788574306919. Disponível em: <http://medcontent.metapress.com/index/A65RM03P4874243N.pdfhttp://books.google.com/books?hl=en&lr=&id=_YRy1lKnniEC&oi=fnd&pg=PA13&dq=História+da+computaç~{a}o:+O+caminho+do+pensamento+e+da+tecnologia&ots=z8c-2wC7f6&sig=0pN4oIx0L6m4PJpVaVX3uVs1pSE>. Citado na página 26.

FRüHWIRTH, T. Logical rules for a lexicographic order constraint solver. *Schrijvers and Frühwirth (2005b)*, 2005. Disponível em: <http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.73.1520&rep=rep1&type=pdf>. Citado na página 19.

Green, Oded and McColl, Robert and Bader, D. A. GPU Merge Path - A GPU Merging Algorithm. In: *Proceedings of the 26th ACM international conference on Supercomputing*.

New York, NY, USA: ACM, 2012. p. 331–340. ISBN 9781450313162. Disponível em: <http://dl.acm.org/citation.cfm?id=2304621>. Citado na página 35.

HEIMLICH, A. A. Desenvolvimento de algoritmos paralelos baseados em GPU para solução de problemas na área nuclear. 2009. Citado na página 26.

HEINZ, S. et al. Burst Tries: A Fast, Efficient Data Structure for String Keys. 2002. Citado na página 46.

HOUAISS, A. (Ed.). *Novo dicionário Folha Webster's:*. São Paulo: Folha da Manhã, 1996. Citado na página 19.

KAKEHI, W. N.; KATSUHIKO. Merging string sequences by longest common prefixes. *IPSJ Digital Courier*, v. 4, 2008. Disponível em: <http://japanlinkcenter.org/JST.JSTAGE/ipsjdc/4.69?from=Google>. Citado na página 21.

KäRKKäINEN, J.; MANZINI, G.; PUGLISI, S. Permuted longest-common-prefix array. *Combinatorial Pattern Matching*, v. 118653, p. 181–192, 2009. Disponível em: <http://link.springer.com/chapter/10.1007/978-3-642-02441-2_17>. Citado na página 22.

KIRK, D. *Programando Para Processadores Paralelos - Uma Abordagem Prática À Programação de Gpu*. [S.l.: s.n.], 2011. 232 p. Citado 5 vezes nas páginas 27, 28, 30, 31 e 32.

KNUTH, D. E. *The Art of Computer Programming, Sorting and Searching*. Redwood City, CA, USA: Addison Wesley Longman Publishing Co., Inc., 1998. 780 p. ISBN 0-201-89685-0. Citado 3 vezes nas páginas 15, 17 e 22.

LEISCHNER, N. GPU algorithms for comparison-based sorting and merging based on multiway selection. 2010. Disponível em: <http://algo2.iti.kit.edu/documents/GPUSortingMerging.pdf>. Citado na página 33.

LEISCHNER, N.; OSIPOV, V.; SANDERS, P. GPU sample sort. *CoRR*, abs/0909.5, p. 1–15, 2009. Disponível em: <http://arxiv.org/abs/0909.5649>. Citado 4 vezes nas páginas 35, 36, 48 e 53.

LIMA, D. S. Estratégia paralela exata para o alinhamento múlltiplo de sequências biológicas utilizando Unidades de Processamento Gráfico (GPU). p. 73, 2012. Disponível em: <http://repositorio.bce.unb.br/handle/10482/13153>. Citado 2 vezes nas páginas 26 e 28.

MCILROY, P. M. M.; BOSTIC, K.; DOUGLAS, M. Engineering radix sort. *Computing systems*, v. 6, p. 5–27, 1993. Disponível em: <http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.22.6990>. Nenhuma citação no texto.

NOBRE, R. H. Paralelismo como solução para redução de complexidade de problemas combinatórios. 2011. Citado 3 vezes nas páginas 26, 27 e 28.

NVIDIA. *NVIDIA® KEPLER GK110 NEXT-GENERATION CUDA® COMPUTE ARCHITECTURE*. 2012. Disponível em: <http://www.nvidia.com/content/PDF/kepler/NV_DS_Tesla_KCompute_Arch_May_2012_LR.pdf>. Citado na página 29.

NVIDIA. *CUDA C Programming Guide*. 2013. Disponível em: <http://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html>. Citado 3 vezes nas páginas 28, 30 e 31.

OWENS, J. D. et al. A Survey of General-Purpose Computation on Graphics Hardware. *Computer Graphics Forum*, v. 26, n. 1, p. 80–113, mar. 2007. ISSN 0167-7055. Disponível em: <http://doi.wiley.com/10.1111/j.1467-8659.2007.01012.x>. Citado na página 28.

Peters, Hagen and Schulz-Hildebrandt, Ole and Luttenberger, N. Fast in-place sorting with cuda based on bitonic sort. In: *Proceedings of the 8th international conference on Parallel processing and applied mathematics: Part I*. Berlin, Heidelberg: Springer-Verlag, 2010. p. 403–410. Disponível em: <http://dl.acm.org/citation.cfm?id=1882792.1882841>. Nenhuma citação no texto.

SATISH, N.; HARRIS, M.; GARLAND, M. Designing efficient sorting algorithms for manycore GPUs. In: *Proceedings of the 2009 IEEE International Symposium on Parallel&Distributed Processing*. Washington, DC, USA: IEEE Computer Society, 2009. (IPDPS '09), p. 1–10. ISBN 978-1-4244-3751-1. Disponível em: <http://dx.doi.org/10.1109/IPDPS.2009.5161005>. Citado 3 vezes nas páginas 12, 36 e 53.

SAYURI, N.; NAKASHIMA, D. Paralelização do modelo BioCrowds para simulação de multidões em GPU e inclusão do efeito de pressão. 2013. Citado na página 26.

SEDGEWICK, R.; WAYNE, K. *Algorithms Fourth Edition*. 4. ed. [S.l.: s.n.], 2011. 976 p. Citado 6 vezes nas páginas 7, 18, 19, 22, 23 e 24.

SEIDEL, R. Data-specific analysis of string sorting. *Proceedings of the Twenty-First Annual ACM-SIAM ...*, p. 1278–1286, 2010. Disponível em: <http://dl.acm.org/citation.cfm?id=1873703>. Citado na página 20.

SINHA, R.; ZOBEL, J. Efficient trie-based sorting of large sets of strings. In: *Proceedings of the 26th Australasian computer ...*. Darlinghurst, Australia, Australia: Australian Computer Society, Inc., 2003. p. 11–18. Disponível em: <http://dl.acm.org/citation.cfm?id=783108>. Citado 2 vezes nas páginas 22 e 38.

SINHA, R.; ZOBEL, J. Cache-conscious sorting of large sets of strings with dynamic tries. *Journal of Experimental Algorithmics (JEA)*, 2004. Disponível em: <http://dl.acm.org/citation.cfm?id=1041517>. Citado na página 25.

SINHA, R.; ZOBEL, J.; RING, D. Cache-efficient string sorting using copying. *Journal of Experimental Algorithmics*, v. 11, n. 1, p. 1.2, fev. 2007. ISSN 10846654. Disponível em: <http://portal.acm.org/citation.cfm?doid=1187436.1187439>. Citado na página 22.

THRUST. *Prefix Sums Function Documentation*. 2012. Disponível em: <http://docs.thrust.googlecode.com/hg/group___prefixsums.html>. Citado na página 42.

VIEIRA, A.; CAMPOI. Comparação de desempenho entre GPGPU e sistemas paralelos. p. 67, 2009. Disponível em: <http://medcontent.metapress.com/index/A65RM03P4874243N.pdf>. Citado 2 vezes nas páginas 26 e 29.

YANG, Q.; DU, Z.; ZHANG, S. Optimized GPU Sorting Algorithms on Special Input Distributions. *2012 11th International Symposium on Distributed Computing and Applications to Business, Engineering & Science*, Ieee, p. 57–61, out. 2012. Disponível em: <http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=6385239>. Citado 2 vezes nas páginas 34 e 53.

ZIVIANI, N. *Projeto de algoritmos: com implementações em Pascal e C*. 3. ed. CENGAGE, 2010. ISBN 9788522110506. Disponível em: <http://books.google.com.br/books?id=kZ-kcQAACAAJ>. Citado 5 vezes nas páginas 12, 15, 16, 18 e 20.