Proposta de problema para implementação em CUDA

Christian Willian Siqueira pires

Resumo—This article presents a sorting algorithm based in the binary representation of elements implemented on CUDA platform, your complexity is O(n*b) where n is the number of elements and b is the count of used bits in the elements representation, the complexity is simplifyed to O(n) because b is a constant, comparing with quicksort, the proposal is faster in amounts of elements greater than 2^{10} elements in 1byte characters sorting, if we consider n CUDA cores, the complexity of the algorithm is O(b).

Index Terms—Computer Society, IEEEtran, journal, paper, O(n), sorting algorithm.

1 INTRODUÇÃO

ALGORITMOS de ordenação são muito usados na área da computação principalmente em armazenamento de dados onde é necessário manter os dados organizados e realizar a inserção e a remoção de dados.

Os algoritmos de ordenação mais comuns como bubblesort, insertion sort, selection sort, quick sort e merge sort, tem ordem de complexidade $O(n^2)$ e $O(n * log_2(n))$. Usar esse tipo de algoritmo para ordenar quantidades massivas de dados pode levar muito tempo, então é comum a pesquisa de novos métodos de ordenação, com intenção de reduzir o tempo de execução.

O algoritmo proposto tem complexidade $O(n * b)$, em que n é a quantidade de elementos e b é a quantidade de bits necessária para a representação dos dados, como b não varia de acordo com a quantidade de elementos, ele se torna um valor constante e assim podemos simplificar a complexidade do algoritmo para $O(n)$.

Neste artigo é apresentado o funcionamento desse algoritmo sendo executado na GPU e comparando seu desenpenho com sua implementação na CPU.

cwsp
Março 03, 2019

2 ALGORITMO

O algoritmo proposto é baseado na representação binária dos números, em que a ordenação ocorre ordenando cada coluna de bits separadamente, colocando os elementos com o bit mais significativo 0 no inicio da lista, e então separando os elementos em dois grupos, em que um dos grupos é formado pelos elementos que tem o bit 0 na coluna analisada e o outro grupo formado pelos elementos que tem o bit 1 na coluna analisada. A operação é repetida para todos os grupos formados. Como podemos ver na figura 1, a cada coluna analisada é formado dois grupos e cada grupo é analisado separadamente formando dois novos grupos,

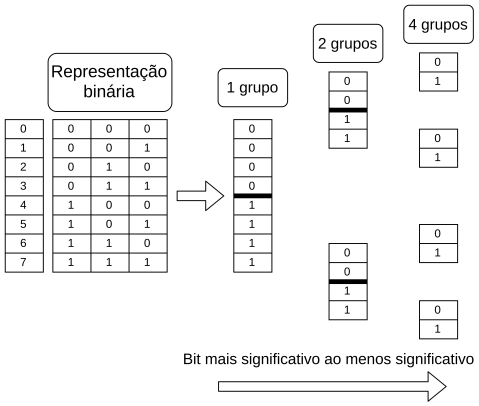

Figura 1. Estrutura dos grupos formados pela recursão do algoritmo proposto.

começando da coluna com o bit mais significativo para a coluna do bit menos significativo.

Dessa forma o algoritmo passa apenas uma vez por cada bit da estrutura de dados totalizando n*b testes, em que n é o número de elementos e b o número de bits da representação dos elementos.

Teoricamente o esse algoritmo se torna mais rápido que o quicksort na ordenação de inteiros de 4 bytes a partir de 2^{32} elementos, a mesma teoria pode ser aplicado na ordenação de qualquer tipo numérico representado em bits, em que a diferença de desempenho sempre ocorre com quantidades maiores que 2^b, onde b é a quantidade de bits do tipo a ser ordenado. No caso de caracteres acima de 2^8 ou 256 elementos o algoritmo proposto é mais rápido.

Sua implementação para GPU, promete um paralelismo dividido em b etapas, se considerarmos que a GPU tem n cores, a complexidade resutante passa a ser O(b), sacrificando um pouco de memória acredito ser possível paralelizar também as b iterações, mas é preciso fazer um estudo mais detalhado se é possível e se vale a pena.

Abaixo podemos ver a implementação do algoritmo proposto na linguagem C usando caracteres como tipo de elementos a serem ordenados.

```
void troca(int *a,int *b)
```

• *C. Pires Graduado em engenharia da computação pela UNIFEI-Campus avançado de Itabira e mestrando na área de visão computacional pela UFF-Instituto de computação.*
E-mail: willian264@hotmail.com

Manuscript received April 19, 2005; revised August 26, 2015.

```
{
    int temp=*a;
    *a=*b;
    *b=temp;
}
void binsort(unsigned char *vet,
         int pos_in,int pos_fin,int bit)
{

    int i, cont=0, tam=pos_fin-pos_in;
    unsigned char mask=(1<<7-bit);

    for(i=pos_in;i<pos_fin;i++)
     if((vet[i]&mask)==0)
       troca(&vet[i],&vet[pos_in+cont++]);

    if(mask!=1)
    if(cont!=0 && cont != tam)
    {
      binsort(vet,pos_in,cont+pos_in,bit+1);
      binsort(vet,cont+pos_in,pos_fin,bit+1);
    }else
    {
      binsort(vet,pos_in,pos_fin,bit+1);
    }
}
```

É usado uma mascara para testar se o bit de determinada coluna é 0 ou 1 usando a operação AND, caso for 0 o elemento é trocado pelo primeiro elemento do grupo, um contador é incrementado e usado para que a próxima troca seja com o segundo elemento e assim sucessivamente, até que todos elementos estejam ordenados de acordo com a coluna de bits em questão.

Uma recursão é usada para criar os dois grupos, em que um é formado apenas pelos elementos com bit da coluna igual a 0 e o outro formado apenas por elementos com bit da coluna igual a 1.

A ultima variável passada na chamada da função indica qual bit da mascara deve ser setado e consequentemente a coluna a ser testada, ela é incrementada a cada chamada da função, avançando dos bits mais significativos aos menos significativos.

Caso a coluna esteja formada apenas por bits igual a 0 ou 1 apenas um grupo é formado com todos elementos.

Para a sua implementação na GPU, uma rápida abordagem seria copiar os dados para uma nova área de memória, evitando problemas comuns de exclusão mutua, más o caso merece um estudo melhor para traçar uma estratégia na estruturação dos dados evitando esses problemas e economizando memória ao mesmo tempo.

3 Resultados na CPU

Para ter resultados corroborando com a justificativa teórica, o algoritmo proposto e o quiscksort foram executados e seu tempo de execução medidos, restritamente na seção do código onde acontece a ordenação.

Foram utilizadas quantidades variadas de elementos na mesma máquina com as mesmas condições de processamento e memória. O desempenho de tempo de execução dos

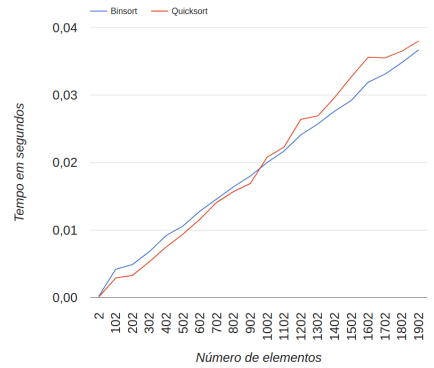

Figura 2. Gráfico do tempo de execução em função do número de elementos dos algoritmos proposto e quicksort até 1902 elementosLINK.

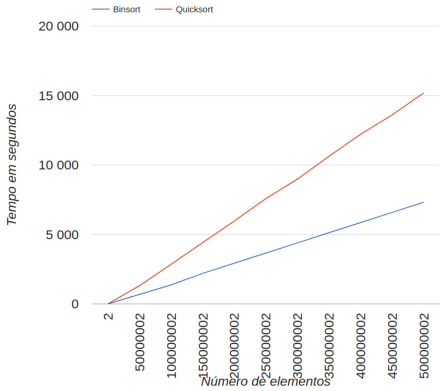

Figura 3. Gráfico do tempo de execução em função do número de elementos dos algoritmos proposto e quicksort até 500 milhões de elementos LINK.

três algoritmos foi anotado e representado em dois gráficos, em que o gráfico da figura 2 tem intenção de demonstrar o ponto de cruzamento do desempenho dos dois algoritmos e o gráfico da 3 tem intenção de demonstrar a linearidade do algoritmo proposto e a discrepância entre os desempenhos dos dois algoritmos.

Para fins de reprodutibilidade do trabalho, no link Repositório, se encontra todos os códigos feitos durante esse estudo bem como as imagens utilizadas nesse artigo e os dados coletados.

O algoritmo proposto em um cenário com muitos elementos se mostrou mais rápido do que o quicksort, principalmente quando o tipo de dados a ser ordenado é pequeno como em caracteres. Quanto menor o tamanho da representação dos bits mais discrepante é a diferença de desempenho, para inteiros de 4 bytes com uma conta simples podemos estimar o ponto de cruzamento das curvas de desempenho em cerca de 16Gb de dados.

Essa característica se torna muito interessante na ordenação de textos, em que a mesma técnica pode ser ampliada e usada para ordenar os caracteres e ainda abstraindo os caracteres como bits ordenando as palavras da mesma forma.

4 Resultados esperados na GPU

Em se falando da comparação entre os dois algoritmos implementados para GPU, é esperado que a comparação com o quiskort, tenha o mesmo efeito, já que o quiscksort

considerando n cores na GPU tem uma complexidade de O(log(n)), o resultado esperado em complexidade para o algoritmo proposto é O(b) onde b é a quantidade de bits da representação numérica utilizada, o que não depende da quantidade de elementos, fazendo as mesmas considerações (n cuda cores), podemos simplificar sua complexidade para O(1), tempo de transferência de memória poderá ser um problema principalmente se todos os dados não couberem na memória ao mesmo tempo, por isso será considerado uma quantidade de dados que a memória da GPU consiga suportar.

REFERÊNCIAS

[1] Yang Y., Yu P. e Gan Y., *Experimental Study on the Five Sort Algorithms*, School of Computer and Information Science - Chongqing Normal University, Chongqing, 400047, China: 978-1-4244-9439-2/11/$26.00 ©2011 IEEE.

Christian Willian Graduado em Engenharia de Computação UNIFEI – Universidade Federal de Itajuba Campus avançado de Itabira - Itabira-MG – Mestrando na área de visão computacional UFF - Universidade Federal Fluminense Instituto de computação - Niterói - RJ