EP2 - Relatório

Bruno B. Scholl
9793586

Victor Seiji Hariki
9793694

Junho 2018

1 Implementação

No nosso EP resolvemos ler todos os dados para um único vetor, armazenando todos os valores em posições adjacentes de memória. Isso facilita a cópia dos dados para a GPU, podendo ser feita em apenas uma transferência de memória; além de facilitar a manipulação do vetor em CUDA - apenas aritmética básica - assim otimizando o uso do *buffer* de memória da GPU.

Queríamos que tal vetor fosse construído de modo que os itens de cada posição da matriz estejam adjacentes um a outro; isto é, se temos três matrizes A, B e C, os valores estão organizados de modo que $A_{1,1}$, $B_{1,1}$ e $C_{1,1}$ estejam um ao lado do outro. Para tal, fizemos a leitura das matrizes de forma não convencional.

A nossa leitura é feita de modo que para uma entrada do seguinte formato:

```
3
***
A B C
D E F
G H I
***
J K L
M N O
P Q R
***
S T U
V W X
Y Z α
```

Temos o vetor:

A	J	S	B	K	T	C	L	U	...	G	P	Y	H	Q	Z	I	R	α

9 itens por matriz $\cdot$ 3 matrizes = 27 valores no vetor

Isso simplifica a redução, pois podemos usar um *kernel* menos complexo que apenas calcula o mínimo de um vetor de tamanho especificado (nesse caso o número de matrizes), e apenas modificar o índice de início do vetor para calcular o mínimo de cada item da matriz separadamente ($A_{1,1}$ começa em 0, $A_{1,2}$ começa em 3, $A_{1,3}$ começa em 6, ...).

Esse vetor é finalmente copiado para a GPU em apenas uma chamada de *cudaMemcpy*, e o processamento dos números começa.

Na gpu, separamos blocos unidimensionais de 1024 *threads* e dividimos o vetor em seções de 2048 itens, dessa forma, cada *threads* tem inicialmente dois valores para comparar.

Para a redução, fizemos dois *kernels*: um deles faz efetivamente a redução, enquanto o outro comprime os valores na memória gerados pela redução.

No *kernel* de redução, cada *thread* compara dois valores, com um espaçamento entre eles igual ao número de *threads* num bloco (igual a 1024 no nosso programa - máximo em CUDA), assim, podemos reduzir 2048 valores em um passo. No início de uma rodada de redução de 2048 itens, todas as *threads* fazem o trabalho de comparação e guardam o resultado na posição do item de menor índice, na segunda iteração, como há apenas metade dos itens a serem comparados, apenas metade das *threads* têm trabalho a fazer; elas, novamente, guardam os resultados no lugar do item de menor índice e assim sucessivamente até restar apenas um valor na posição 0 da memória compartilhada do bloco que é copiado para a posição correspondente ao bloco na memória global.

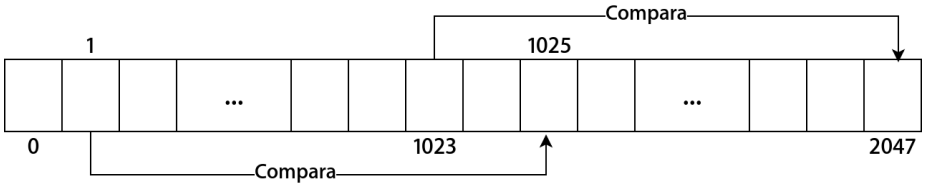

Figure 1: Na figura a *thread* 1 compara os valores de 1 e $1025 = 1 + 1024$ e a *thread* 1023 compara os valores 1023 e $2047 = 1023 + 1024$

O *kernel* de compressão é usado para duas funções importantes que fazem, no fundo, a mesma coisa: a aglutinação de valores originalmente espalhados, mas igualmente espaçados no vetor. Esse *kernel* faz a simples tarefa de pegar valores em intervalos regulares do vetor e comprimi-los para o começo do vetor. Um exemplo, dado um vetor v e um intervalo de tamanho 6, seria a cópia feita pela definição $v[i] = v[i * 6]$.

Essa capacidade é usada de duas formas no processo. A primeira é que, se o vetor excede 2048 matrizes, temos que usar múltiplos blocos para a redução completa do vetor. Mas há a restrição de que cada bloco pode gravar apenas na sua área delimitada de memória global no fim de sua execução (Caso contrário, poderia interferir com a leitura de outro bloco).

Então cada bloco salva o seu resultado no local do primeiro item que ele leu da memória global. Assim, os resultados ficam espalhados, um no 0, no 2048, no 4096..., e, como há múltiplos resultados, uma segunda iteração de redução é requerida. Para essa segunda execução, é necessário que todos os valores estejam contíguos no começo do vetor. E para isso é executada a compressão.

E, no fim, uma última compressão é executada para levar os valores finais de cada elemento para o começo do vetor, que estão no começo das áreas para cada elemento (0, número de matrizes, número de matrizes * 2, ...), no fim, permitindo a transferência do resultado da GPU para a memória de modo fácil.

2 Performance

A performance da versão de CUDA foi notavelmente maior que a da versão executada em apenas uma *thread* na CPU. Mas o maior limitante da velocidade do programa foi a leitura dos dados a partir do disco que levou, em média, 15 vezes mais tempo do que o processamento das matrizes de forma sequencial. Para 50.000.000 matrizes, por exemplo, nossa execução demorou 46 segundos para a leitura do arquivo; 3 segundos para a execução da versão serial; e 0.6 segundo para a versão de CUDA, o que equivale a apenas $\frac{1}{5}$ do tempo sequencial.

3 Conclusão

Percebendo a insignificância do tempo de processamento se comparada com a leitura do arquivo, não há uma diferença substancial na execução da redução, tomando em conta o carregamento dos dados do disco. Então, em conclusão, a implementação do processo em CUDA, no contexto desse exercício, não mostrou uma melhora, pelo menos de forma perceptível. A diferença seria mais óbvia caso mais matrizes fossem usadas, mas ainda seria pequena em termos de tempo total de execução. Agora, em outros possíveis casos de uso, como em uma situação onde não é necessário carregar tantos dados do disco, e sim processar dados que já estão na memória ou são gerados durante a execução, a vantagem da utilização de aceleradores como GPUs(CUDA, no caso) é clara.