T3-Computação Paralela com GPUs

Alunos:
Luiz Henrique Murback Wiedmer - GRR20221234
Eduardo Giehl - GRR20221222

1.Introdução

Esse documento relata a otimização do trabalho 2 de computação paralela com GPUs em CUDA, que consiste em 5 kernels usados em conjunto para ordenar um vetor. Será explicada as alterações feitas nos kernels, mostrando a comparação de testes e seus resultados.

2.Kernel BlocksHistoAndScan

A versão inicial do algoritmo do kernel blockAndGlobalHisto inicia calculando o intervalo de valores entre min e max, o menor e o maior valor presentes em input, e o dividindo em h faixas de mesmo tamanho, salvando o limite superior de cada faixa em um vetor limite. Então faz uso deste vetor limite para determinar quantos elementos de input, acessados por esse bloco, estão presentes em cada uma das faixas. Cada bloco executa esses passos preenchendo uma linha de hh, gerando os histogramas por blocos. Então hg é preenchido com a soma das linhas de hh, gerando o histograma global.

Já a versão otimizada remove completamente a necessidade do vetor limite ao calcular dinamicamente a qual faixa o elemento avaliado pertence e já o contabilizando à faixa correta.

3.Kernel GlobalHistoScan

O algoritmo do kernel GlobalHistoScan começa com cada thread escrevendo um valor hg no vetor de shared memory XY, e completando o vetor com 0s caso h(o tamanho do histograma global) não seja potência de 2. Após isso, é executado um algoritmo de scan baseado no algoritmo de Blelloch[1]. A ideia se baseia em realizar a soma em forma de “árvore”, primeiramente construindo soma parciais nos nós internos da árvore. Após isso, as “árvore” é percorrida novamente construindo todos os valores de saída a partir das somas parciais. Após isso, os valores em XY são escritos por cada thread com índice menor que h no vetor global shg, para garantir que apena valores reais, e não 0s do padding, sejam escritos na memória global.

4.Kernel VerticalScanHH

O kernel VerticalScanHH funciona com a mesma teoria do GlobalHistoScan, mas ao invés de ser feito sobre apenas um vetro de tamanho h, ele é executado sobre uma matriz de tamanho nb X h(sendo nb o número de blocos lançado pelos kernels BlocksHistoAndScan e Partition, e h o tamanho do histograma global), assim gerando uma matriz do mesmo tamanho com o scan de cada coluna de HH. Para fazer isso, primeiramente, cada bloco escreve uma coluna no vetor de shared memory XY, assim cada bloco de threads pode lidar com uma coluna como se fosse um vetor, usando o algoritmo de Blelloch[1]. Ao final do processo, os valores são escritos no matriz global psv, invertendo o processo feito na escrita, então cada bloco, escreve uma coluna da matriz.

5.Kernel Partition

A versão inicial do algoritmo do kernel partitionKernel começa com cada bloco calculando o vetor hlsh, histograma por linha em shared, através da soma de hg com uma linha de hh referente ao bloco que está executando. Como resultado, cada posição de hlsh contém o índice de output onde começa cada faixa de valores para o bloco que o calculou. Então calcula o intervalo de valores entre min e max, o menor e o maior valor presentes em input, e o divide em h faixas de mesmo tamanho, salvando o limite superior de cada faixa em um vetor limite Após isso, utiliza limite para identificar a qual faixa cada elemento de input, acessado pelo bloco, pertence, então insere o elemento em output na posição correspondente indicada por hlsh e atualiza hlsh para não sobrescrever o elemento inserido.

Já a versão otimizada remove completamente a necessidade do vetor limite ao calcular dinamicamente a qual faixa o elemento de input avaliado pertence.

6.Kernel segmentedBitonicSort

O kernel segmentedBitonicSort começa com cada thread copiando alguns valores do vetor global de input para o vetor de shared memory s_key, além de que cada thread pode completar o vetor de shared memory com UINT_MAX, no caso de posições de padding, adicionadas unicamente para garantir a funcionalidade do algoritmo. Após isso é iniciado um for, em cada iteração dele, é construída uma sequência ordenada maior, fazendo 2, 4, 8, etc. Para isso é utilizado um outro loop for interno, que vai dizer o stride entre os dois elementos que cada thread deve comparar e possivelmente trocar. Como queremos que uma thread consiga lidar com mais de 2 elementos, é necessário mais um loop, que inícia em threadIdx, e vai crescendo até a metade do tamanho do vetor, sempre somando o tamanho do bloco. Com isso é possível simular mais threads, permitindo que mesmo que sejam usadas menos de tamanho/2 threads, o algoritmo ainda funcione. Dentro desse for é feita a comparação citada anteriormente. Essa comparação inicialmente é feita de maneira que metade é ascendente(troca se a > b), e a outra metade é descendente(troca se a <= b). Após esses dois loops aninhados, que geram uma sequência bitônica (metade ordenada de um jeito, metade de outro), é necessário fazer um loop final para formar uma única sequência ordenada. A lógica desse loop é a mesma do for interno discutido anteriormente,inclusive necessitando a parte da “simulação” de threads, com a diferença que todas as comparações são feitas na direção correta(nesse caso ascendente). Após tudo isso, é realizado a cópia de s_key para o vetor global d_Dst_key, levando sempre em conta o padding.

7.Conjunto final

Com esses 5 kernels, é possível fazer um algoritmo de ordenação seguindo os seguintes passos:

1 – Use BlocksHistoAndScan para gerar a lista de histogramas e o histograma global;

2 – Calcule o scan do histograma global com GlobalHistoScan;

3 – Calcule o scan da lista de histogramas com VerticalScanHH;

4 – Utilize o que foi calculado até agora no Partition para inserir os elementos de input na faixa correta do vetor de output;

5 – Ordene o vetor geral lançando h blocos, com cada bloco ordenando uma das h divisões do vetor output.

8.Testes

Os kernels foram testados com vetores de tamanho 1 milhão, 2 milhões, 4 milhões e 8 milhões preenchidos pseudo-aleatoriamente com unsigned ints, e sempre com h=1024 e cada teste repetido 10 vezes. Foi contado o tempo médio de execução de cada teste, e com isso foi calculada a vazão em GE/s. Além disso, para comparação, foram feitos os mesmos testes no kernel de ordenação da biblioteca Thrust. Todos os testes foram feitos na máquina NV00, que tem uma GPU NVIDIA GeForce GTX 1080 Ti.

Os resultados alcançados são apresentados na tabela a seguir:

Tamanho	Vazao_mppSort (GE/s)	Vazao_Thrust (GE/s)	Speedup(Slowdown)
1M	1,500182	1,355801	1,106491
2M	1,450904	2,302009	0,630277
4M	1,387055	2,770144	0,500716
8M	1,379744	3,299359	0,418186

É possível verificar que Thrust se saí melhor em 3 dos 4 casos, além de ter grandes diferenças de desempenho de acordo com o tamanho da entrada, diferentemente do mppSort, que tem vazão estável.

Também foi testado o algoritmo segmentedBitonicSort diretamente contra o Thrust, os resultados foram os seguintes:

Tamanho	Vazao_mppSort (GE/s)	Vazao_Thrust (GE/s)	Speedup(Slowdown)
1M	3,31038	1,34513	2,46102
2M	3,4386	1,98274	1,73426
4M	3,47531	2,55645	1,35943
8M	3,49553	3,49553	1,12933

É possível fazer a mesma verificação feita pelos resultados do mppSort, o bitonicSort tem vazão estável de acordo com o tamanho da entrada, ao passo que o Thrust ganha desempenho de acordo com o aumento do tamanho da entrada.

9.Referências

[1] https://developer.nvidia.com/gpugems/gpugems3/part-vi-gpu-computing/chapter-39-parallel-prefix-sum-scan-cuda