TRABALHO 1 DE COMPUTAÇÃO PARALELA EM GPU

Prof. Wagner Zola

Luiz Fernando Giongo dos Santos

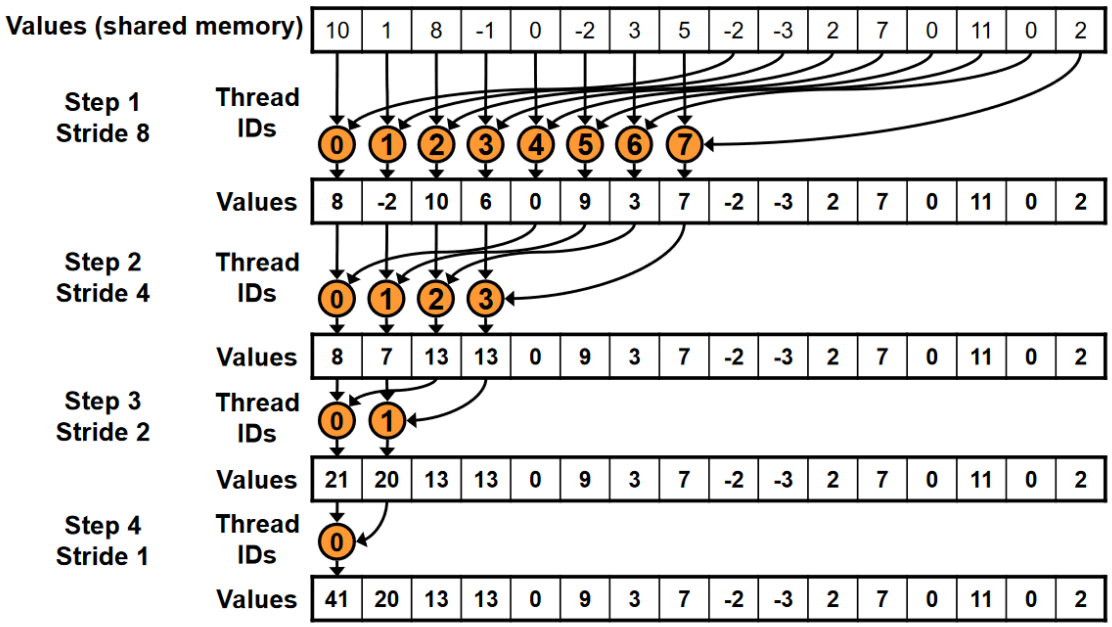
Muriki Gusmão Yamanaka

Introdução breve:

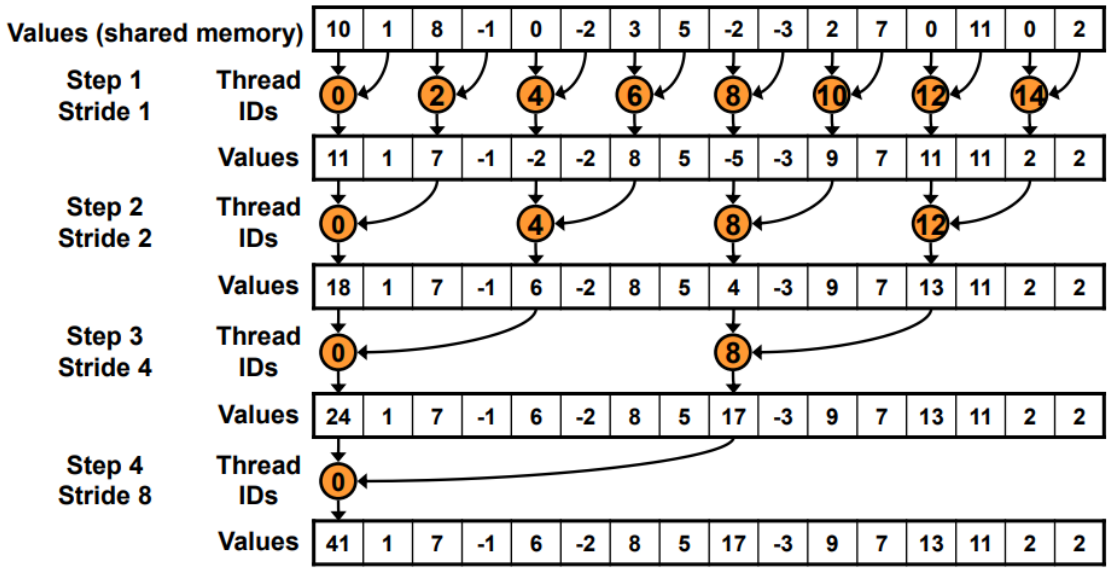
O trabalho é responsável por demonstrar formas distintas de se fazer um reduce Max, uma forma utilizando operações comuns e acabando com operações atômicas. Outro apenas utilizando operações atômicas e o ultimo utilizando o método da lib thrust.

Esse projeto serve para demonstrar a utilidade e eficiência do thrust, ao mesmo tempo que desenvolvemos as habilidades necessárias para desenvolver código para GPU's usando CUDA, nesse caso, a típica implementação do "Reduce".

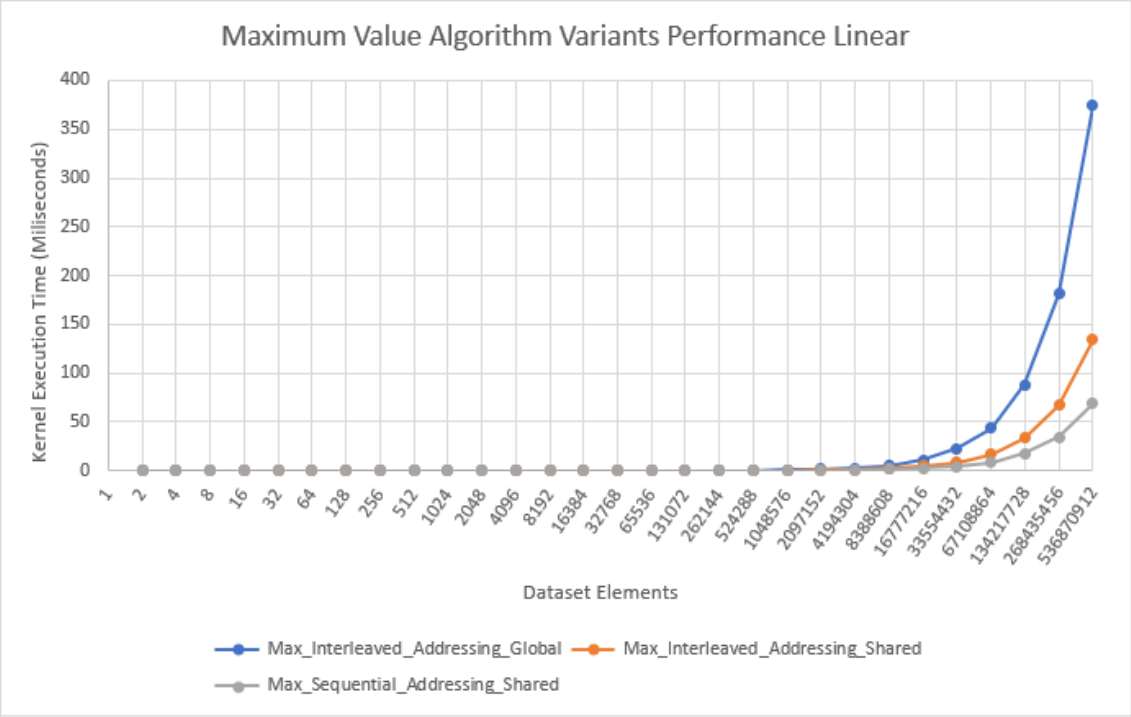
Existem 2 vertentes ou versões do reduce que podem ser feitas, a sequencial e a interleaved. Para esse trabalho decidimos optar pelo interleaved, não foi uma escolha por algum motivo em particular, apenas decidimos no começo do trabalho, antes de saber das vantagens do sequencial, e preferimos manter assim.



(exemplo de sequencial)



(Exemplo de interleaved)



(Gráfico da comparação de suas velocidades retirado de https://github.com/MaxKotlan/Cuda-Find-Max-Using-Parallel-Reduction)

Embora o reduce sequencial seja levemente mais rápido, todas as formas apresentam velocidade exponencial para valores extremamente altos, logo nada que apresente uma notável diferença para o nosso projeto.

Acreditamos que o sequencial tenha essa vantagem principalmente pela memoria coalescida, que é melhor obtida em uma função sequencial de alta proximidade local da memória.

Estrutura de arquivos:

O nosso código foi quase que inteiramente feito em um único arquivo, pelo único motivo de ser exclusivamente mais simples de programar, importamos 2 arquivos .c, o Chronos, usado para temporizar o nosso processo, e Log, usado para reportar erros do código.

Nossa main e códigos de GPU foram implementados no cudaReduceMax.Cu, esse possui todas as versões de reduce e da função thrust que usamos para os testes.

O runScript.sh pode ser utilizado para automaticamente rodar o Makefile e os 4 testes que decidimos implementar, 2 desses obrigatórios a mando do professor.

Organização do código:

A função atômica foi baseada em uma versão encontrada no “stackoverflow”, seu funcionamento é bem simples, utilizando o AtomicMax para int, e transformando nossos valores float em int e novamente em float para a resposta.

Temos 2 funções reduceMax, uma sendo normalmente comparada e a outra usando apenas operações atômicas. Como pedido, a função reduceMax não atômico ainda precisa utilizar atômico na sua sequência de comparações final, sendo assim um mix dos 2 métodos.

Nossos “Reduces” foram em parte inspirados na logica de um projeto no Git(https://github.com/MaxKotlan/Cuda-Find-Max-Using-Parallel-Reduction), porém apenas o seu funcionamento do loop interno, sendo que o código escolhido não era de blocos permanentes, logo necessitou de uma grande quantidade de alterações para ficar de acordo com o projeto.

Dados e resultados:

Teste	Elementos	Repetições	Reduce	Reduce Atomic	Thrust
Teste0	1000	30	1,390 mS	1,143 mS	1,312 mS
Teste1	1000000	30	17,298 mS	13,216 mS	1,979 mS
Teste2	16000000	30	78,403 mS	37,006 mS	6,485 mS
Teste3	40000000	30	97,707 mS	68,203 mS	14,288 mS

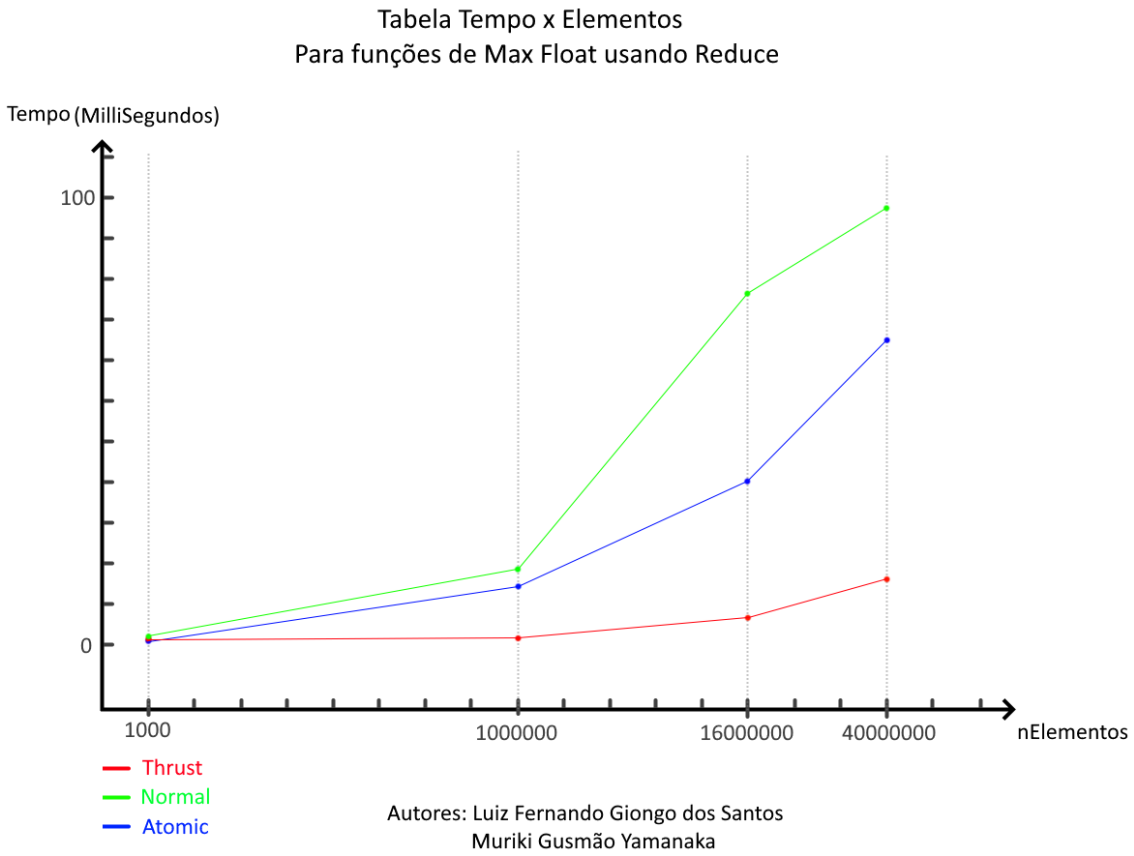
Autores: Muriki Gusmão Yamanaka
Luiz Fernando Giongo dos Santos

Fizemos 4 testes para o nosso simples código, 2 desses obrigatórios pelo professor, marcados com vermelho.

Inicialmente esses dados apresentaram um resultado estranho, sendo que o Atomic, por apresentar uma logica mais complexa e uma segurança maior nas suas operações, deveria ter o maior tempo de processo. Mas entre as nossas funções, ele apresentou o menor tempo.

Eu acredito que esse pode ter sido o caso por culpa da função atômica, muitas GPU’s devem dar suporte nativo a essas operações e possuir Hardware próprio para resolvê-los, talvez acelerando o processo do Kernel.

Esses resultados foram então grafados para uma mais fácil legibilidade.



É importante ressaltar que embora o tempo esteja constantemente espaçado, o horizontal da tabela, isso é, o numero de elementos, aumenta em uma proporção exponencial.

Isso explica, por exemplo, a distorção não esperada do tempo do Kernel Reduce sem Atomic, chamado de Normal na tabela.

Essa tabela novamente, repete o nosso entender do Reduce, Thrust se mantem o melhor Kernel com até 1 nível de grandeza, isso é claro nos nossos testes, acreditamos que se continuássemos com testes cada vez maiores, iremos continuar a ver a progressão exponencial dos Kernels, e o Thrust tendo a menor exponencialidade, iria cada vez mais ser superior aos seus concorrentes.

Conclusão:

A livraria Thrust é otimizada e trabalhada a anos e anos, um código mal elaborado como os nossos Reduces jamais conseguiria concorrer a algo tão otimizado quanto o Thrust.

Ainda assim, é contra intuitivo o fato que o Kernel com funções Atomicas apresenta um tempo melhor ao Kernel que utiliza operações comuns, novamente atribuímos isso a talvez otimizações de hardware que a GPU possa ter para suporte nativo de funções Atomicas.

De qualquer forma, todos as funções são exponenciais, apenas com graus de exponencialidade diferentes. Oque significa que ele rapidamente se tornaria inviável para vetores extremamente grandes.