Relatório do Trabalho Final de Programação com GPU

Bruno Belenda Meurer - 114083265
Guilherme de Brito Freire - 114049740

Implementações Propostas

Implementação mais simples possível

Começamos o trabalho fazendo uma implementação inicial em GPU bem rudimentar, sem o uso de nenhum artifício mais avançado de memória, apenas para checar como a GPU se sairia caso algum programador quisesse tirar proveito da capacidade de paralelismo dela sem necessariamente passar por todo o trabalho de otimização necessário para atingir os mais altos níveis de desempenho.

Essa implementação simplesmente transferia as informações da memória principal para a memória global da GPU e realiza os cálculos com blocos de threads em duas dimensões (a dimensão dos blocos poderia ser determinada antes de iniciar o programa).

Obs: Todos os testes foram realizados em uma Nvidia GTX 1080 (A não ser que seja indicado o contrário) e os programas sequenciais foram compilados com a flag de otimização -O3.

Implementação com memória constante

O próximo passo no nosso trabalho foi tentar arranjar alguma maneira de incorporar os diferentes tipos de memória que conhecemos ao longo do período para tentarmos tirar mais proveito do paralelismo oferecido pela GPU. Dos 3 tipos de memória específicos que aprendemos, memória compartilhada, constante e de textura, resolvemos fazer uso da memória constante pois era a que tinha uma utilidade mais aparente logo de imediato.

Memória constante é utilizada para acelerar o acesso a variáveis que, como o nome já diz, não se alteram durante a execução do programa. Nesse programa temos alguns exemplos bem óbvios, como as dimensões da matriz de entrada e as variáveis h1 e h2, e alguns um pouco menos óbvios como transformar a equação:

$$\frac{2+h_1 a_{ij}}{4 \cdot (1 + \frac{h_1^2}{h_2^2})} \text{ em } \frac{2}{denominador} + \frac{h_1 a_{ij}}{denominador}$$

onde denominador = $4 \cdot (1 + \frac{h_1^2}{h_2^2})$.

Dessa maneira, denominador e $\frac{2}{denominador}$, podem ser calculados apenas uma vez na cpu e depois colocados na memória constante evitando fazer várias contas por iteração no programa.

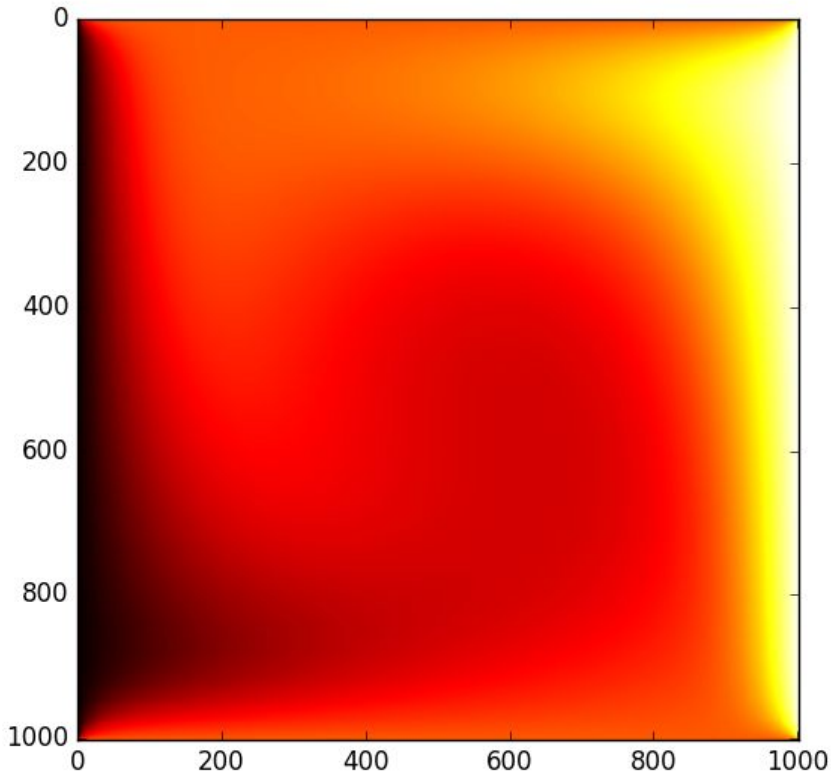


Gráfico mostrando a matriz estabilizada gerada em paralelo.

Implementação com memória compartilhada

Outra maneira de resolver o problema com um ganho muito bom de velocidade é possível usando o artifício de memória compartilhada. Para isso, adaptamos a ideia do uso dessa memória a partir do exercício de multiplicação de matrizes feito ao longo do curso.

A ideia consiste em copiar pequenos pedaços da matriz localizada na memória global (proporcional ao tamanho do bloco) e armazenar na memória compartilhada deste bloco. Como existem múltiplos acessos dentro de um bloco aos mesmos elementos da matriz, encontramos um pontecial ganho a ser explorado. Uma mesma posição da matriz, pode ser usada até quatro vezes em uma mesma iteração. Como a busca desse valor foi modificado para usar a memo compartilhada, o tempo diminui consideravelmente. Então em princípio, essa modificação seria extremamente boa.

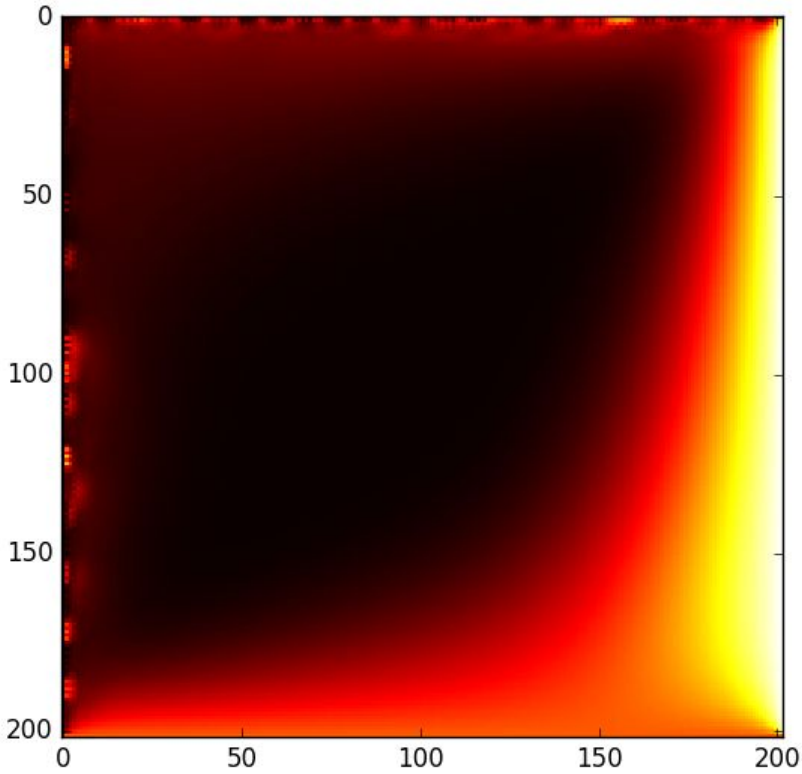
Entretanto, a implementação dessa tática se provou um desafio. A primeira abordagem consistia em fazer as threads que estavam esperando a sua vez de calcular (quando a iteração era da cor oposta) copiarem seus valores para a memória

compartilhada do bloco, de forma que as threads que estão na vez de calcular não precisem fazer esse trabalho. Entretanto, em primeira instância, tentamos alocar apenas o necessário de memória - que seria efetivamente usada no cálculo - e trabalhar com os índices das threads e da matriz. A conversão, entretanto não se mostrou simples, principalmente levando em consideração matrizes de não quadradas e principalmente com número ímpar de elementos.

Partimos, então, para a segunda tentativa. Alocar uma matriz com dimensões maiores do que o bloco para incluir a borda exterior. Dessa forma, o cálculo com índices foi enormemente simplificado. Apesar de termos conseguido bolar um algoritmo em teoria é no papel sobre como resolver o cálculo usando essas matrizes menores com memória compartilhada, montar essas matrizes se tornou um problema.

Passamos grande parte do tempo tentando entender as razões pelas quais a matriz não se comportava da forma esperada em diversas situações. Usamos inclusive ferramentas como o cuda-gdb e cuda-memcheck para analisar a memória e, portanto, ver o conteúdo dessas matrizes compartilhadas.

Por fim, passamos para a última abordagem desse problema. Nela, separamos as threads em “interiores” e “exteriores”. As threads interiores são aquelas que usam apenas elementos da matriz dentro do seu bloco para calcular seu valor. As matrizes exteriores, usam pelo menos um elemento que está fora do bloco para calcular seu valor (incluindo as bordas fixas). Assim, calculamos as threads interiores com a memória compartilhada e as exteriores com a memória global. Essa tática foi a que mais deu certo dentre todos os testes, mas mesmo assim não conseguimos resultados corretos para a solução do problema descrito no enunciado do trabalho.



Melhor resultado que obtivemos usando memória compartilhada.

Tentativas experimentais

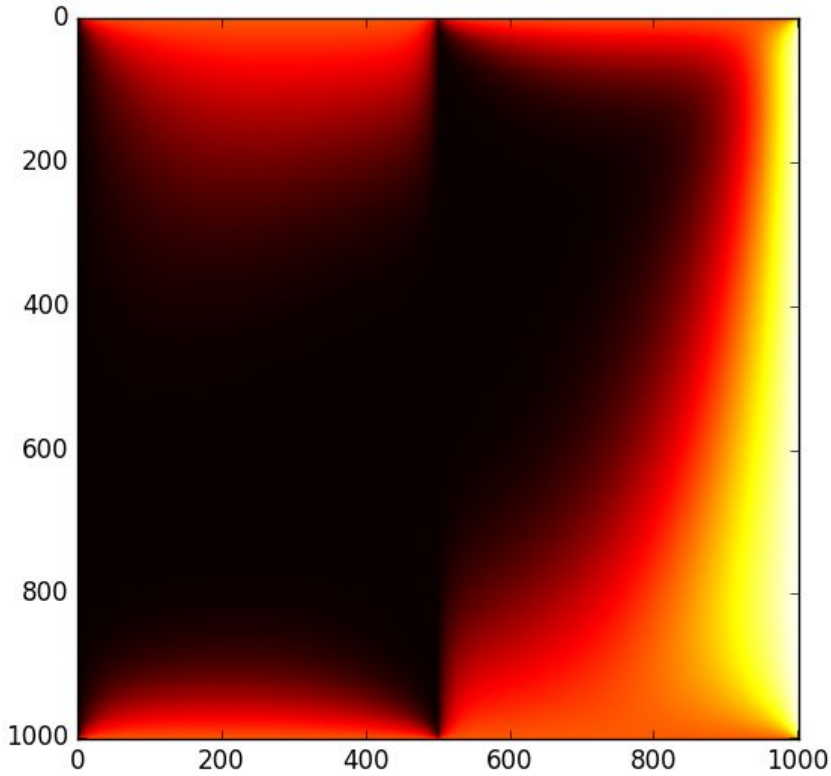
Nesta seção vamos falar a respeito de tentativas de implementação do código em CUDA fazendo uso de coisas que não foram ensinadas durante o curso mas que pensamos que poderiam aumentar ainda mais o speed up.

Múltiplas GPU's

Uma das primeiras ideias que nos veio à cabeça foi a adição de uma segunda GPU ao problema de modo que ela pudesse acelerar ainda mais os cálculos, dividindo o trabalho todo entre elas. Utilizando esse conceito, a primeira tentativa de implementação de um código desse tipo envolveu tentar dividir a carga igualmente entre as duas GPU's.

Apesar de ser um bom começo para vários problemas, essa tentativa não serviu para esse problema em particular devido ao fato de que uma GPU não consegue ver o que a outra fez na sua metade da matriz sem que as duas fiquem fazendo transferências

de memória bastante custosas. O resultado fica mais ou menos assim:



Dá para ver nitidamente a linha que separa as áreas das duas GPU's. Isso acontece porque é como se cada uma das fronteiras que se encontram nessa linha tivessem o valor inicial sempre, que, nesse caso, era 0.

Ao nos depararmos com isso, começamos a explorar diferentes maneiras de contornar esse erro. Uma delas foi a mencionada acima de fazer transferências dos resultados de uma para a outra. Isso se tornou inviável, na prática, pois acabava demorando mais do que usar uma única placa.

A melhor alternativa que pudemos encontrar foi a de utlizar a tecnologia de peer to peer (P2P) entre as placas. Resumidamente, ela permitiria que as placas tivessem acesso às memórias uma da outra e pudessem realizar as atualizações conforme necessário sem precisar ter que recorrer às custosas transferências entre memórias. Infelizmente não foi possível implementar uma solução dessa maneira, pois as placas que tínhamos à nossa disposição não eram capazes de fazer uso dessa tecnologia entre elas.

Memória de superfície

A memória de textura, à qual fomos apresentados em classe, apesar de um grande potencial devido à sua especialidade em aproveitar a localidade espacial de dados em

2D, não serve para esse problema devido ao fato de não poder ser atualizada depois que já foi escrita na memória. Como as matrizes são atualizadas a cada iteração, essa memória não serve para esse problema.

Depois de um pouco de pesquisa, porém, acabamos encontrando um outro tipo de memória de textura chamado de memória de superfície. Essa memória permite que sejam feitas escritas a ela pelo kernel. Isso a torna uma ótima candidata para esse problema.

Para aproveitar melhor o conceito de localidade espacial desse tipo de memória, uma ideia de implementação que nos surgiu foi a de separar a matriz principal em duas submatrizes (uma dos pontos vermelhos e outra dos pontos azuis) de modo que as duas pudessem ser transferidas para a memória de superfície. Como todos os pontos estariam lado a lado, isso possibilitaria o máximo de desempenho da cache dessa memória. Isso aliado a outros tipos de memória da GPU permitiriam teoricamente um desempenho superior, aliado à flexibilidade de poder alterar a matriz ao longo da execução do programa.

Apesar desse potencial todo, a memória de superfície não é trivial de ser usada. Requer certos cuidados e entendimentos mais avançados sobre o CUDA. Com o tempo que tivemos, não conseguimos estudar esse tipo de memória o suficiente para aplicá-la ao problema.

Extras:

Além da implementação do trabalho, também criamos ferramentas com o intuito de melhorar a visualização do programa.

Primeiro, fizemos um script em python que mostra um gráfico “heatmap” da matriz. O programa lê a entrada de um arquivo chamado *sample.txt* e cria o gráfico a partir dele. Esse programa foi muito útil para debug e entendimento do trabalho.

Um problema que estávamos tendo com essa primeira versão é que ela requeria o término da execução para criar a matriz. Seria bom poder ver como a matriz ia evoluindo com as iterações.

Pensando nisso, criamos uma segunda versão da visualização, uma versão ao vivo. Nessa versão, o programa desenha a matriz, da mesma forma que o anterior, e além disso, atualiza os valores da matriz a cada 500ms. Instruções de como rodar o programa com a visualização ao vivo estão no github do trabalho. É muito interessante - diria até hipnotizante - assistir a matriz evoluir com as iterações. Se mudarmos as funções *a* e *b* do problema original, inclusive, conseguimos animações surpreendentes.