Paralelismo em GPU's para cálculo de estêncil de diferenças centradas em função do tempo

Christian Willian S. Pires
Engenheiro da Computação Mestrando em Computação na área de Computaçao visual
UFF - Universidade Federal Fluminense – Campus Praia Vermelha
Niterói-RJ, Brasil
Email: cwspires@id.uff.br

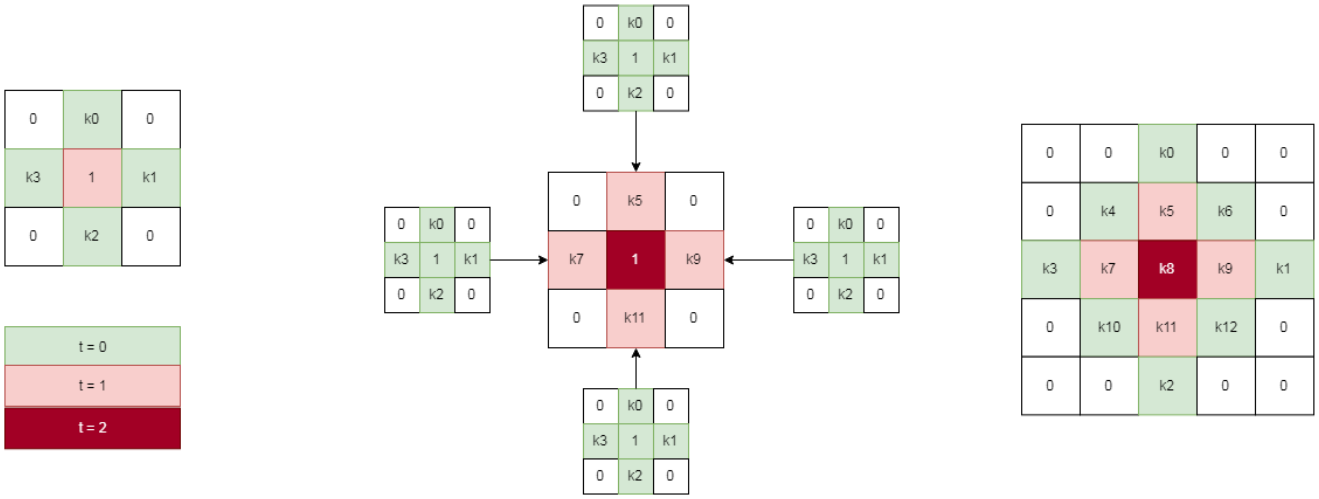

Figura 1. Crescimento da memória necessária para o cálculo de estêncil de diferenças centradas do instante t=0 até o instante t=2

***Resumo*—Dado uma matriz bidimensional com informações de um determinado instante de tempo t, deseja-se calcular o resultado no instante t+n, para uma determinada discretização de tempo. Parar realizar esta operação é necessário utilizar trechos de memória, que, na forma convencional da organização de dados não se encontram próximos uns dos outros, o que impede um aproveitamento razoável dos recursos de armazenamento de memória compartilhada por SM's (stream multiprocessor) e do barramento de memória.**

***Palavras-Chave*-estêncil de diferenças centradas; CUDA; Memória compartilhada; Programação paralela;**

I. Problema

Para cada elemento da matriz bidimensional recebida como entrada, é necessário calcular o valor do instante de tempo seguinte, esse cálculo depende dos valores em uma vizinhança 4 conectada, esses 4 valores são multiplicados por um conjunto de constantes definidas, semelhante a todos os elementos definida na figura 1 como kn.

Na programação de GPU's, o acesso a memória é o gargalo da maioria das aplicações, principalmente, quando é preciso realizar acessos em uma ordem aleatória ou desordenada. No problema de estêncil de diferenças centradas, os dados necessários para o cálculo de cada elemento, não se encontram originalmente em sequencia na memória.

Na GPU, temos disponível, entre outros recursos, dois tipos de memória, a global e a compartilhada por SM, em que a memória global é muito mais lenta que a memória compartilhada, em contrapartida, tem muito menos capacidade de armazenamento. O objetivo é organizar os dados, e determinar uma melhor quantidade de instantes de tempo a serem calculados dentro do mesmo Warp, aproveitando ao máximo os recursos do barramento e de memória compartilhada por SM.

Quando é necessário calcular mais de um instante de tempo a memória utilizada por cada thread aumenta exponencialmente, e sua organização também deve ser alterada, para atender a nova demanda, como pode ser visto na figura 1. É preciso calcular esse numero de iterações e testar diferentes abordagens, que possam otimizar o tempo de execução do algoritmo, fazendo proveito dos recursos de memória disponíveis.

II. Paralelismo

Por se tratar de um problema onde existe uma dependência limitada de dados por cada elemento, e o resultado final de

cada elemento em uma determinada iteração é independente dos resultados dos demais elementos. É possível que cada cálculo de instante de tempo de cada elemento sejam executados simultaneamente em um instante de tempo.

Mais importante do que isso no caso do paralelismo de GPU's é que para garantir um paralelismo real, algumas características devem ser observadas, como por exemplo, nesse caso, as operações executadas para cada elemento são idênticas variando apenas os dados utilizados, isso é muito bom para a arquitetura utilizada na GPU, em que, cada SM, tem apenas um schreduler que é responsável por transmitir aos cores, as operações a serem realizadas. Evitando assim que algumas threads fiquem ociosas como acontece em casos de divergencia de threads que entram em caminhos de execução diferentes.

III. Algoritmo

Os dados são colocados na memória da GPU pelo Host e então para cada n iterações a função calcula na GPU é chamada novamente.

Para cada chamada do host ao device pela função calcula, a memória correspondente aos elementos é copiada para a região compartilhada de memória

O algoritmo multiplica cada valor do kernel representado na figura 1 e soma os resultados armazenando na mesma região de memória allocada no espaço compartilhado.

Todos elementos tem uma thread associada que executam a mesma operação n vezes, em que n, define a quantidade de iterações para o mesmo carregamento de memória da global para a compartilhada.

Calculado as n iterações, os novos valores são levados para a memória global aguardando a nova chamada de calcula que é um novo conjunto de blocos para calcular as n iterações subsequentes.

É preciso sincronizar a execução entre as chamadas de calcula, para garantir que os dados que vão para a memoria compartilhada são da iteração correta.

```
__GLOBAL__ calcula(double* dados,
                double * kernel,int n)
{
    shared = cudaMallocShared(NElements);

    cudaMemSharedCopy(global,shared,
                Elements,GlobalToShared);

while(n>0)
{
soma=0;
soma += kernel[k3]*shared[ID_Elemento-1x];
soma += kernel[k0]*shared[ID_Elemento+1x];
soma += kernel[k1]*shared[ID_Elemento-1y];
soma += kernel[k2]*shared[ID_Elemento+1y];
soma += shared[ID_Elemento];
shared[ID_Elemento] = soma;
n--;
}
    cudaMemSharedCopy(global,shared,
                Elements,SharedToGlobal);

'   cudaFreeShared(shared);
}
int main(){
    H_dados = mallloc();
    D_dados = cudaMalloc();
    cudaMenCopy(H_dados,D_dados,
            tam,HostToDevice);

    for(int i=0;i<iteracoes;i+=n)
    {
        CudaExecute<<calcula(D_dados,
                    kernel,n)>>><tam+i>>;
    }

    cudaMenCopy(H_dados,D_dados,
            tam,DeviceToHost);
}
```

Código 1: Pseudo código da implementação do algoritmo para o cálculo do estêncil de diferenças centradas bidimensional.

IV. Resultados esperados

É esperado que o overhead gerado por acessos desordenados a memória e a diferença de desempenho entre a memória global e compartilhada, tenha um custo de desempenho superior, ao de calcular uma quantidade de operações resultantes de uma equação em função de n(numero de iterações por grid). Relacionado a quantidade de cálculos realizados que não será aproveitado pelo próximo warp.

Gerando assim uma redução do tempo de processamento de um determinado conjunto de dados, em um determinado instante de tempo.

V. Conclusão

Neste projeto podemos experimentar uma forma interessante de melhorar o desempenho de aplicações sendo executadas em GPU, trocando uma maior quantidade de processamento nos cores por menos transações de memória. Se validada, a técnica pode ser aplicada em problemas semelhantes.

Também é possível melhorar essa implementação, reduzindo o overload das chamadas de kernel utilizando a técnica de stride.