Paralelização de Autômatos Celulares em Placas Gráficas com CUDA

Marcos Paulo Riccioni de Melos [1]

[1]Departamento de Ciência da Computação – Instituto Multidisciplinar – Universidade Federal Rural do Rio de Janeiro (UFRRJ)
Nova Iguaçu – RJ – Brasil

marcospaulo.r.melos@gmail.com

Abstract. *Cellular automata is a technique for solving problems containing elements that interact with their immediate vicinity and requiring external data instance. The efficiency of the algorithm was tested in several scenarios and compared between the sequential and parallel approaches. The results show that the methodology used to improve the performance problems of this nature, may be improved by using the shared memory of the GPU. There is also a limitation on the size of the problem in the parallel implementation with this distribution in more plates and more computers would be a more accurate solution.*

Resumo. *Autômatos celulares é uma técnica de solução de problemas contendo elementos que interagem com sua vizinhança próxima e necessitam de dados externos a instância. A eficiência do algoritmo foi testada em diversos cenários e comparadas entre as abordagens sequencial e paralela. Os resultados mostram que a metodologia usada melhora o desempenho para problemas dessa natureza, podendo ser melhorada com o uso da memória compartilhada da GPU. Também há uma limitação no tamanho do problema na implementação paralela, com isso a distribuição em mais placas e mais computadores seria uma solução mais acertada.*

1. Introdução

A abordagem de autômatos celulares consiste em uma grelha infinita e regular de células, cada uma podendo estar em um número finito de estados, que variam de acordo com regras determinantes. Esta técnica se adapta muito bem a problemas que discretizam populações e simulam a interação extracelular das mesmas, criando vínculos de vizinhança que são atualizados a cada interação de tempo na população [SCHIFF 2008].

O tempo também é discretizado, onde o estado de uma célula no tempo t é obtido em função do estado no tempo t-1, levando em consideração seu estado e os estados de células vizinhas. Esta vizinhança corresponde a células próximas que obedecem a uma formação pré-definida. Geralmente, os problemas são discretizados em forma de matrizes, no qual cada elemento é uma ou mais posições da matriz [SCHIFF 2008; GREMONINI & VICENTINI 2008].

[GARDNER 1970] e [SCHIFF 2008] apresentam a aplicação teórica Gol (Game of Life). A mesma reproduz, através de regras simples, as alterações e mudanças em

grupos de seres vivos, ilustrando o comportamento de células em relação ao tempo e seu ambiente.

O Gol apresenta apenas quatro regras definidas por [Gardner 1970].

O custo para calcular os estados das células e obter as alterações e mudanças de grupo pode ficar alto, dependendo do tamanho da matriz e da quantidade de gerações a serem simuladas. O objetivo deste trabalho é melhorar o desempenho através da paralelização massiva dos passos da aplicação. Para tanto, foi desenvolvida uma versão paralela do GoL, denominada GolCU, para aproveitar o paralelismo disponível nas placas gráficas (GPU – Graphics Processor Unit) através da linguagem CUDA (Compute Unified Device Architecture), que devido a sua natureza SIMD (Single Instruction Multiple Data) mostra-se muito adequada a problemas desta classe, onde obtém um ganho significativo em performance frente a processadores convencionais, pois fornece quantidades superiores de unidades de processamento [EICHENBERGER et al., ____; SANDERS & KANDROT 2010].

2. Metodologia

A solução paralela abordada foi utilizando apenas uma GPU. Outras abordagens também podem ser criadas, tendo em vista a natureza do problema.

A aplicação GoL é inerentemente paralela e muito adequada para executar em GPU, já que é explícito no tempo. A principal característica da GPU é processar a mesma instrução sobre diferentes dados, logo, cada core da GPU pode executar uma célula da matriz utilizada na simulação do GoL. Assim, na abordagem proposta, os dados são copiados para a memória global da GPU e cada thread calcula paralelamente o estado de sua célula correspondente em um instante de tempo. O resultado final da simulação é copiado de volta para CPU para exibição da simulação.

A quantidade de threads criadas depende do tamanho do problema e da quantidade de cores disponíveis na GPU. Quando o tamanho da aplicação, ou seja, quando a quantidade de threads ou células é maior do que a quantidade de cores disponíveis, o processamento continua sendo feito em paralelo, mas é criada uma fila para o processamento das células excedentes. Além disso, outra característica da aplicação é que a cada passo, é necessário a sincronização das células para que todas as threads reiniciem o processamento com o dado atualizado.

No caso ideal para a implementação apresentada e o ambiente de execução disponível, o tamanho máximo da matriz de entrada seria de aproximadamente 8191x8191, ou um total de 67.107.840 elementos, de acordo com a capacidade máxima da placa utilizada. Essa capacidade máxima é definida pela quantidade de blocos e quantidade máxima de threads por bloco disponíveis na GPU.

3. Implementação

Utilizando o Visual Studio 2010, a implementação foi baseada na linguagem C/C++, utilizando o CUDA como arquitetura de computação paralela.

A parte lógica do algoritmo Gol é mantida para todas as abordagens sugeridas. Por ser uma aplicação teórica, não há entrada nem saída esperada correta, apenas a correta computação até a última interação do tempo.

3.1. Gol Sequencial

A seguir é exemplificada a parte principal do código do *Gol* sequencial:

```
repita para i menor que geracao {
        repita para j menor que tamX
            repita para k menor que tamY {
                    regras do Gol;
            }
}
```

Como visto, essa abordagem requer laços de repetição aninhados, o que ocasiona grande gasto de tempo apenas para as atualizações das matrizes.

3.2. Gol paralelo com 1 GPU

A seguir são exemplificados alguns trechos do código do Gol paralelo:

```
__global__ void calcViz(const int h_tam, int *d_matrix, int *d_aux){
        regras do Gol;
        __syncthreads();
}
int main(){
        repita i menor que geracao {
            calcViz <<<dimGrid, dimBlock>>> (h_tam, d_matrix, d_aux);
            cudaStatus = cudaDeviceSynchronize();
        }
}
```

Como dito anteriormente, nesta abordagem todos os dados são transferidos para a GPU, que calcula cada geração de modo paralelo dentro da placa. Cada thread é responsável pelo cálculo do estado de uma célula.

É passado para a função do Device o tamanho do vetor, o vetor com os dados iniciais da simulação, além de um vetor auxiliar. Cada célula é identificada pelo identificador único da thread, nessa divisão cada thread realiza os cálculos referente a célula correspondente do vetor de entrada. Após, os dados são atualizados no vetor principal, para serem usados na próxima interação.

5. Resultados e Discussão

Para avaliar a proposta foi utilizado o seguinte ambiente: processador Intel Core i7 3770, com clock de 3,4GHz, 8mb de memória cache e 12Gb de memória RAM a 1333MHz, 2 placas de vídeo NVIDIA GT 640, Ubuntu 12.04, CUDA 6.

Os resultados são apresentados na Tabela 1, onde foram utilizados valores médios a partir de cinco tomadas de tempo. Os tempos das implementações foram obtidos utilizando a função clock() da linguagem C, onde é medido o tempo de toda a implementação e na implementação paralela foi usada cudaEventRecord().

O sistema operacional foi colocado em modo texto e nenhum outro processo foi inicializado.

Table 1. Tempos medidos

Tamanho da Matriz	Quantidade de de Gerações	Tempo médio sequencial (ms)	Desvio padrão tempo sequencial(ms)	Tempo médio paralelo (ms)	Desvio padrão tempo paralelo(ms)	Speed-Up
10	1000	10,000	0,000	21,207	0,776	0,4715
100	1000	480,000	0,000	170,221	0,801	2,8198
1000	1000	48.260.000,000	5,477	13.474.863,281	18,114	3,5814
2000	1000	192.430.000,000	11,401	53.914.535,156	50,182	3,5691
5000	1000	1.199.970.000,000	91,268	336.979.875,000	64,591	3,5609
8000	1000	3.072.020.000,000	179,220	864.370.187,500	65,320	3,5540
10000	1000	4.800.080.000,000	208,989	1.352.506.500,000	76,621	3,5490

O speed-up foi utilizado como medida de comparação entre os tempos de execução sequencial e paralelo [HENNESSY & PATTERSON 2011]. Foi observado que o tempo paralelo em relação a matriz quadrada de tamanho 10 é pior que o sequencial, isso é explicado pela necessidade de troca de informações entre o Host e Device. Quando aumenta-se o tamanho da matriz a perda inicial com a comunicação é absorvida pelo tempo da computação, com isso o desempenho do sequencial é facilmente ultrapassado.

6. Conclusão

A partir dos resultados foi possível observar que, problemas resolvidos por autômatos celulares se beneficiam com a abordagem apresentada. A implementação paralela desenvolvida pode ser melhorada em com a utilização da memória compartilhada. A limitação de tamanho pode ser aumentada repartindo os dados para cálculos em diversos computadores, com diversas GPU's ou atribuindo a computação de mais células a cada thread.

7. Referências

Gardner M. (1970). The fantastic combinations of John Conway's new solitaire game "life". Scientific American, v. 223, n. 4, p. 120-123.

Schiff J. L. (2008) Cellular Automata: A discrete View of the World. Wiley Interscience, New Jersey.

Sanders J. and Kandrot E. (2010) CUDA by Example: an introduction to general-purpose GPU programming. Addison-Wesley Professional.

Eichenberger A. E., Wu P. and O'Brien K. Vectorization for SIMD Architectures with Alignment Constraints. IBM T.J. Watson Research Center, Yorktown Heights, NY. Disponível em < http://researcher.watson.ibm.com/researcher/files/us-alexe/paper-eichen-pldi04.pdf >. Acesso em julho de 2014.

Gremonini L. and Vicentini E. (2008) Autômatos celulares: revisão bibliográfica e exemplos de implementações. Revista Eletrônica Lato Sensu – UNICENTRO, ed. 6.

Hennessy J. L. and Patterson D. A. (2011) Computer Architecture: A quantitative Approach. Morgan Kaufman, 5th ed.