Métodos em Física Computacional
Lattice Gas

L. Q. Costa Campos

25 de Fevereiro de 2013

"The symmetry gods are benevolent"
– U. Frisch

1 Modelo

1.1 Autômatos Celulares

Para termos um autômato celular, duas coisas são necessárias: uma coleção de sítios e um conjunto de regras. Cada sítio pode estar em um número finito de estados, e é em geral uma rede regular. O conjunto de regras nos diz como um certo sítio vai estar no próximo passo de tempo, dado o estado atual do sítio e comumente dos seus vizinhos. Para um estudo aprofundado de autômatos celulares, veja [1].

1.2 Lattice Gas

Neste projeto, foi utilizado o modelo criado por Frisch, Hassslacher e Pomeau [2], em 1986. Neste modelo, nosso automato está localizado em uma rede triangular regular de lado L. Em nossa abstração, dizemos que cada sítio é uma localização no espaço real ligado a seis outros, fazendo uma rede triangular regular. Essas ligações são chamadas de arestas (usando nomenclatura de grafos) ou canais. Cada um desses canais pode conter uma partícula. O sítio pode também estar ocupado por uma partícula parada ou por uma barreira. Então, cada sítio pode se encontrar em um dos 256+1 possiveis estados. As barreiras descrevem o contorno do sistema.

Nossas regras têm que seguir as leis de conservação de massa e de momento. Por exemplo, na Fig. 1 (a), um dos possiveis resultados poderia ser (b), mas não (c), pois este resultado não conserva momento, nem (d), por que a massa não seria conservada. O conjunto completo de regras está exposto no livro-texto do curso [3].

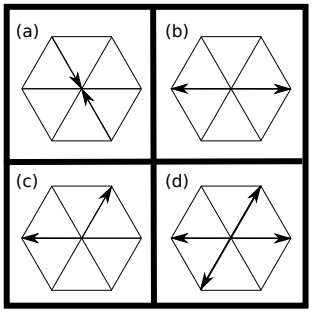

Figura 1: Exemplo de possíveis resultados para a condição apresentada em (a). Destas, somente (b) é válida, pois (c) e (d) não conservam momento e massa, respectivamente.

É importante dizer que a rede triangular regular é a única rede bidimensional que exibe simetria o suficiente para tender às equações de Navier-Stokes[4]. A rede quadrada não exibe tal grau de simetra. Também não existem redes tridimensionais que satisfaçam todos os requisitos para que tenhamos as equação de Navier-Stokes. Para utilizar este modelo em 3D, é necessário usar redes de dimensão superior e então as projetar em três dimensões [5].

Nome e Direção	Valor Decimal	Valor Binário
EMPTY	0	00000000
RIGHT	1	00000001
RIGHT_DOWN	2	00000010
LEFT_DOWN	4	00000100
LEFT	8	00001000
LEFT_UP	16	00010000
RIGHT_UP	32	00100000
STATIONARY	64	01000000
BARRIER	128	10000000

Tabela 1: Tabela com valores e significados dos bits utilizados. Esta representação vale para máquinas little endian.

2 Implementação

2.1 Implementação Serial

Implementamos um programa em C++ para simular o modelo descrito na Sec. 1. Utilizamos também OpenGL, através do GLFW para a visualização em tempo real do nosso sistema. Nele, quanto mais vermelho um ponto, maior a densidade dele. Como dito na Sec. 1.2, cada sítio pode estar em apenas uma quantidade finita de estados, sendo cada um desses estados uma combinação da presença de partículas vindo em cada direção do lattice ou com barreira. E em cada canal, pode haver somente uma partícula.

Tomamos proveito desta condição para implementar cada sítio usando um número inteiro. Sabemos que em um computador todo número, inclusive os inteiros, são uma combinação de vários bits, em ordem. Na maioria das máquinas atuais, cada número do tipo int é composto por 32 bits, donde utilizaremos apenas os oito primeiros deles. A cada um destes oito bits damos um significado, como presença de uma particula de gás vindo pela esquerda, ou a existência de partículas paradas. A tabela 1 mostra o significado de cada um dos bits, assim como o nome pelos o qual são chamados no programa e seu valor na base decimal.

Podemos então combinar as diversas situações utilizando as máscaras bit-a-bit | e & (para mais informações sobre estes, vide Apêndice A). Então, um sítio com uma particula indo para a direita e uma partícula parada pode ser construido com RIGHT | LEFT. Podemos testar a presença de um certo bit usando o operador &. Por exemplo, (RIGHT | LEFT) & RIGHT == RIGHT.

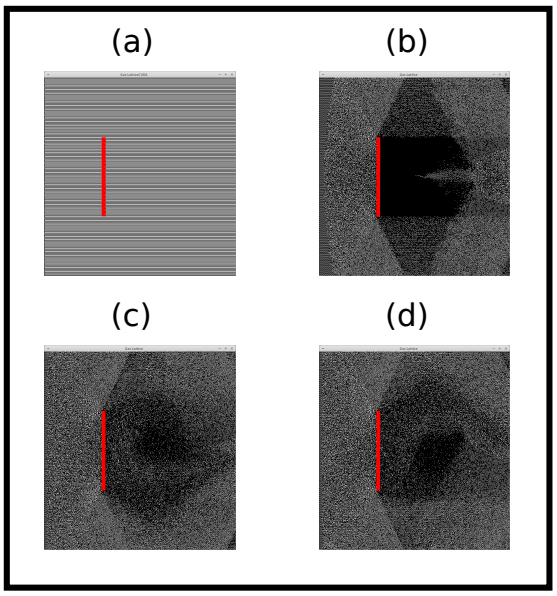

Figura 2: Exemplo de sistema com tamanho 384× 384 sítios, em serial.

O array de regras foi criado de tal forma que o valor correspondente a um certo índice é exatamente o resultado na próxima iteração do valor deste índice. Por exemplo, se em um passo t o sítio s_{ij}^t se encontra no estado RIGHT_DOWN | LEFT | RIGHT, então no próximo passo ele deverá ter um valor s_{ij}^{t+1} = RIGHT_UP | LEFT_DOWN | RIGHT_DOWN (vide [3], pg. 573). O nosso array de regras, chamado rules, é construido de forma que rules [RIGHT_DOWN | LEFT | RIGHT] = RIGHT_UP | LEFT_DOWN | RIGHT_DOWN. Essa propriedade de rules é válida para todos os estados. A dinâmica é realizada ao atribuirmos a cada sítio o valor indexado pelo seu estado atual. A vantagem de utilizar tal esquema é que os operadores bit-a-bit são automaticamente paralelos, e pela construção de rules, não são necessários condicionais na maior parte do programa.

Neste programa foram criadas duas classes de

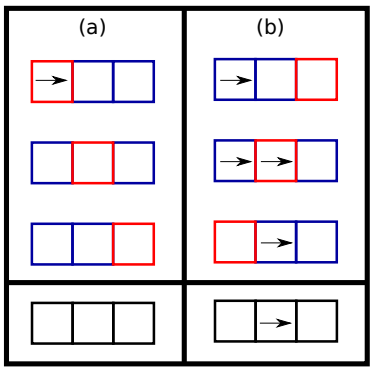

Figura 3: Exemplo incorreto de iteração em uma rede unidimensional utilizando somente um array para os sítios. O quadrado vermelho denota o sítio que está sendo atualizado atualmente, e o azul os sítios que estão sendo lidos para atualizar o sítio atual. Em (a) iteramos da esquerda para a direita e em (b) em da direita para a esquerda.

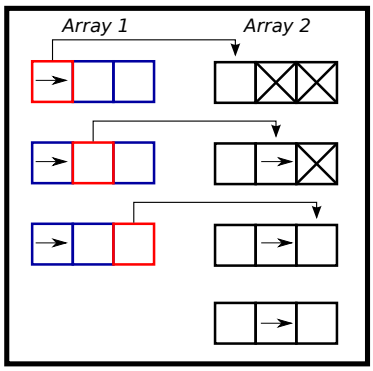

Figura 4: Exemplo de iteração em uma rede unidimensional utilizando dois arrays para os sítios. O array da esqueda mostra o valores atuais do sítio, e o da direita, os valores no próximo passo de tempo. No array da direita, os sítios com um X ainda não foram atualizados. De forma semelhante à Fig. 3, o quadrado vermelho mostra o sítio que está sendo atualizado, e o azul os que estão sendo lidos para atualizar o sítio atual.

C++, Cell, que descreve um único sítio, e Lattice que descreve o sistema completo. A classe Cell contém o número de vizinhos que está conectado a cada sítio, sua posição (x,y), e seu estado atual. Ele inclui também funções para ser atualizado, cálculo do momento linear e cálculo da densidade.

A classe Lattice é a nossa classe principal de simulação. Nela estão contidas as funções essenciais para a simulação e os arrays que contém os sítios. São necessários dois arrays distintos para realizar a simulação, por que é necessário atualizar toda rede paralelamente. Se utilizassemos somente uma rede, introduziriamos erros na simulação, já que teriamos uma dependência da ordem que iteramos sobre os sistema.

A Fig. 3 mostra um sistema fictício unidimensional de apenas três sítios durante uma única atualização no tempo. A condição inicial é a mesma em ambos, com uma partícula indo para a direita no primeiro sítio. Em Fig. 3 (a), os sítios são atualizados da esquerda para a direita, com o sítio que está sendo iterado em vermelho. Ao iterarmos o primeiro sítio, lemos tanto a primeiro quanto o último sítio, e como nenhum dos dois têm partículas vindo em direção ao sítio atual, marcamos este como vazio. Percebemos então que perdemos a única partícula da rede, e todos os resultados seguintes serão vazios.

Na Fig. 3(b), os sítios são iterados da esquerda para a direita então iteramos primeiramente o último sítio. Observamos o segundo e primeiro sítios e vemos que não há nenhuma partícula em nossa direção. Marcamos o último sítio como vazio e continuamos nossa jornada, indo para o segundo sítio. Ao lermos o primeiro, percebemos que finalmente existe uma partícula vindo em nossa direção, com velocidade para a direita, e que no último sítio não temos nenhuma partícula. Marcamos então o segundo sítio como contendo uma partícula indo para a direita. Como não há nenhuma partícula vindo para o primeiro sítio, o marcamos como vazio. Podemos ver então que os resultados dos dois esquemas de atualização difere sensivelmente, mesmo para um caso simples em uma dimensão.

Se utilizarmos dois array não teremos mais o problema da dependência com a ordem que iteramos sobre o lattice. A Fig. 4 mostra o mesmo sistema que

a Fig. 3, mas utilizando um esquema de array duplo. Vamos percorrer o Array 1 em ordem crescente. Os vizinhos da primeira célula são vazios, então ele também deverá ser vazio. Gravamos então essa informação no Array 2. O próximo sítio recebe um partícula com velocidade para a direita do sítio primeiro, e nenhuma do último sítio. Então, colocamos no Array 2 uma partícula indo para a direita no segundo sítio. Finalmente, nenhum vizinho do último sítio fornece particulas, deixando-o vazio. Escrevemos essa última informação no Array 2 e completamos a iteração. Como pode ser visto, o resultado está correto. Teriamos este mesmo resultado se a ordem que percorremos o Array 1 fosse inversa, já que Array 1 é constante durante este passo de tempo. No próximo passo de tempo, o Array 1 e o Array 2 vão inverter suas funções, i. e., percorremos o Array 2 e escreveremos o resultado no Array 1.

A classe Lattice lida com qual array deve ser atualizado a cada iteração de forma automática. É importante perceber que, por questões de eficiência, todos os arrays são unidimensionais. Para que estes arrays tenham a mesma topologia da rede triangular, utilizamos o algoritmo descrito em [6], Cap. 13.

Lattice realiza a dinâmica utilizando a função FullUpdate(), que chama UpdateLattice1() e UpdateLattice2(), em sequência. Esta classe também apresenta funções para calcular o momento total do sistema, bem como sua massa. A velocidade do programa se encontra na Sec. 2.3, comparando a implementação atual com a paralela. Um snapshot do sistema pode ser encontrado em Fig. 2.

2.2 Implementação Paralela

Paralelizamos o programa da Sec. 2.1, utilizando a plataforma da nVidia, CUDA. Por causa do pobre suporte de CUDA a objetos, a classe Cell foi banida. A classe Lattice foi aumentada para incorporar algumas funções de Cell. Ela ainda ainda vai ser utilizada, pois não a usamos na placa de vídeo, somente na CPU. Lattice agora chama funções externas, que serão rodadas na placa de vídeo. Esse novo tipo de função é chamado de kernel. Cada kernel é lançado automaticamente em todos os processadores da placa de vídeo, possivelmente com mais de uma instância por

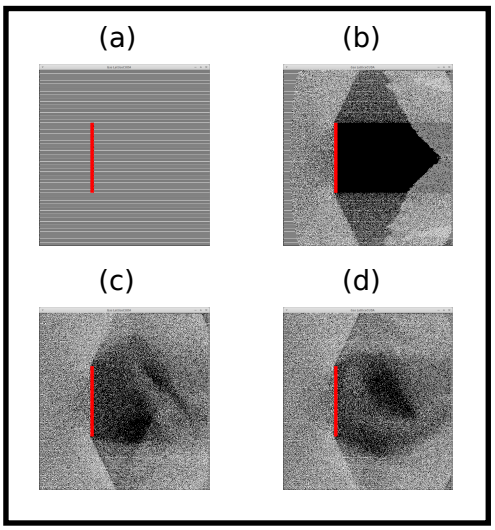

Figura 5: Exemplo de sistema com tamanho 384× 384 sítios, em serial. Foram utilizados 192 blocos de 768 threads cada.

processador. Cada uma dessas instâncias é chamada de uma thread, e forma a unidade lógica mínima.

As threads se organizam em blocos, que por sua vez constituem o grid. Cada bloco está associado a um e somente um multiprocessador da placa de vídeo. Uma GPU pode ser formada por mais de um multiprocessador. Por exemplo, a GT 630M é composta por 2 multiprocessadores (MP), cada qual formado por 48 processadores (SP). No nosso sistema, nós utilizamos uma thread por sítio da rede. Ao usar CUDA, é mais vantajoso minimizar a quantidade de passagens de memória da GPU para a CPU e viceversa, pois essa é uma operação custosa considerando a baixa velocidade relativa do barramento PCI.

O kernel que é responsável por atualizar a rede se chama UpdateCells(). Dentro dele, cada um dos sítios é atualizado paralelamente. Para não haver qualquer forma de data races, ainda se faz necessário usar dois arrays distintos. Nesta implementação, utilizamos também VBOs[1][2] por ser muito mais rápido

[1] Mais informações sobre VBOs em [7], Cap. 15

[2] A partir da versão 3.1 do OpenGL, as operações glBegin() e glEnd() foram *banidas*, e agora a única forma de se desenhar

L	Serial (s)	ms/ite	CUDA(s)	ms/ite
192	93.746	0.2397	11.977	0.05864
384	363.928	1.8964	44.495	0.22248
576	845.127	4.2256	96.055	0.48028
768	1478.969	7.3948	168.134	0.84067

Tabela 2: Velocidade dos programas implementados nesse projeto. As colunas 2 e 5 mostram a velocidade total do programa, enquanto as 3 e 5 mostram o tempo por iteração. Cada programa iterarou 2×10^5 vezes. Os testes seriam foram realizados em um núcleo do processador Intel(R) Core(TM) i7-3612QM CPU @ 2.10GHz, e os de CUDA foram calculados em uma GTX630M. Os programas foram compilados com gcc 4.6.3 e nvcc 5.0. Eles rodaram em um Ubuntu 12.04 64bits.

que desenhar os nossos resultados sítio por sítio, além de VBOs apresentarem interoperabilidade com CUDA, evitando assim uma custosa passagem de memória da placa de vídeo para o processador. De fato, em todo o programa, há apenas um momento de trânsferência de memória entre a GPU e a CPU, no inicio. Após essa inicialização, todo os dados necessários se encontram disponíveis na GPU. Um instântaneo de tal programa se encontra em Fig. 5.

2.3 Benchmark

Realizamos testes para sistemas de vários tamanhos, variando de $L = 192$ a $L = 768$, onde L é o tamanho lateral do sistema. Iteramos cada programa 2×10^5 vezes, sem visuzalização. Encontramos os resultados resumidos na Tabela 2.

Como pode ser observado, os programas rodaram bastante rápido, gastando apenas 7ms nos caso mais lento. Considerando que ainda existe margem para otimização do sistema, os resultados se mostram promissores. Os programas em CUDA foram razoavelmente mais rápidos que os seriais. De forma similar, deixamos possibilidade de otimização do sistema paralelo para os interessados.

é usando VBOs e Vertex Array. Ainda assim, a maioria das empresas de hardware garantiram que vão continuar a aceitar a forma antiga de glBegin/glEnd.

A Operadores bit-a-bit

Neste apêndice vamos mostrar o uso dos operadores bit-a-bit disponíveis em C++. Para nossos exemplos, utilizaremos inteiros de 8 bits.

C++ nos provê seis desses operadores,

- |, ou
- &, e
- ^ , xor, ou ou exclusivo
- <<, deslocar para a esqueda
- >>, deslocar para a direita
- ~, complementar

Vamos mostrar o uso de cada um deles. Eles funcionam exatamente da forma que podemos esperar dos nomes deles, mas em cada bit independentemente.

A.1 operadoradores |, & e ^

O operador | atua como o operador lógico || em cada bit. Veja como exemplo a seguinte operação

```
   0001 0011
|  0001 1010
--------------
   0001 1011
```

Não é surpreendente então que o operador & atue da mesma maneira que &&. Se considerarmos os mesmos números da operação anterior,

```
   0001 0011
&  0001 1010
--------------
   0001 0010
```

Finalizando estes operadores mais simples, temos o operador "ou exclusivo", ^ . O resultado dele é somente verdadeiro se ambos os argumentos forem diferentes. Os conjuntos de entradas (0,0) e (1,1) serão ambos falsos, enquanto (1,0) e (0,1) serão verdadeiros. Aproveitando nosso exemplo anterior,

```
   0001 0011
^  0001 1010
--------------
   0000 1001
```

A.2 operadores deslocamento

C/C++ nos dá dois operadores de deslocamento, um para a esquerda, << e outro para a direita, >>. Apliquemos então esses dois operadores aos nossos exemplos-padrão.

```
   0001 0011
<< 2
--------------
   0100 1100

   0001 0011
>> 2
--------------
   0000 0100
```

É importante notar que esses operadores não são nem associativos nem comutativos. O resultado também difere dependendo da ordem que operarmos. Finalmente, um não é o inverso perfeito do outro. Ou seja, existem números i e j tal que $((i << j) >> j) \neq i$! Isso ocorre por que bits que saem dos limites da representação são exterminados, e os que aparecem no outro extremo são sempre zero.

Como observação final, vale dizer que existem dois outros operadores comumemente usados, mas que não são disponibilizados pelo C++, rotate left e rotate right. Eles se assemelham a << e >>, mas os dígitos que saem em um extremo reaparecem em outro. Para os aventureiros, eles estão disponíveis no assembly de quase todos os processadores. Para os processadores de arquitetura x86, temos os comandos ROL (Rotate Left) e ROR (Rotate Right).

A.3 operador ~

Por fim, temos o operador negação. Ele é o mais simples dos operadores. Ele é unário, ou seja, só recebe um argumento. Ele simplesmente inverte cada bit do número, como visto a seguir.

```
 ~ 0001 0011
--------------
   1110 1100
```

Referências

[1] Stephen Wolfram. Statistical mechanics of cellular automata. *Rev. Mod. Phys.*, 55:601–644, Jul 1983.

[2] U. Frisch, B. Hasslacher, and Y. Pomeau. Lattice-gas automata for the navier-stokes equation. *Phys. Rev. Lett.*, 56:1505–1508, Apr 1986.

[3] Harvey Gould, Jan Tobochnik, and Christian Wolfgang. *An Introduction to Computer Simulation Methods: Applications to Physical Systems (3rd Edition).* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2005.

[4] Daniel H Rothman and Stéphane Zaleski. *Lattice-Gas Cellular Automata: Simple Models of Complex Hydrodynamics.* Cambridge Univ. Press, Cambridge, 1997.

[5] U. Frisch, D. d'Humières, B. Hasslacher, P. Lallemand, Y. Pomeau, and J.P. Rivet. Lattice gas hydrodynamics in two and three dimensions. *Complex Systems*, 1, 1987.

[6] M. E. J. Newman and G. T. Barkema. *Monte Carlo Methods in Statistical Physics.* Oxford University Press, USA, April 1999.

[7] Richard Wright, Benjamin Lipchak, and Nicholas Haemel. *Opengl superbible: comprehensive tutorial and reference, fourth edition.* Addison-Wesley Professional, fourth edition, 2007.