Implementação paralela de autômatos celulares para CPUs multi-core e GPUs CUDA

Bryan Lincoln Marques de Oliveira
Computação Paralela 2021.1 - PPGCC - Universidade Federal de Goiás

***Abstract*—Este estudo investiga o desempenho da implementação de autômatos celulares em CPUs multi-core e GPUs CUDA, considerando diferentes configurações de grade, raio de vizinhança e processo de renderização. O objetivo é avaliar o potencial de processamento paralelo dessa abordagem para simulações complexas. O Jogo da Vida de Conway foi escolhido como prova de conceito para avaliar os tempos de processamento e speedups obtidos. Os resultados mostraram que a implementação em GPU foi capaz de alcançar um speedup de até 627 vezes em relação à implementação sequencial, enquanto a implementação em CPU multi-core alcançou um speedup máximo de 7,5 vezes. Esses resultados indicam que a implementação de autômatos celulares em processadores paralelos é uma técnica promissora para acelerar aplicações em diversas áreas, como modelagem de fluidos e simulação de sistemas biológicos. O *framework* de código aberto disponibilizado pode ser utilizado como ponto de partida para outras aplicações interessadas em implementar autômatos celulares em processamento paralelo.**

***Index Terms*—Autômatos celulares, jogo da vida, CUDA, multi-core, computação paralela.**

I. Introdução

Os recentes avanços nas tecnologias de manufatura de componentes eletrônicos e de unidades de computação permitiram o barateamento das mesmas e, ao mesmo tempo, tornaram possível a construção de supercomputadores que alcançam a marca dos petaFLOPS de poder de processamento. Parte desse poder se deve ao eficiente design de seus circuitos e processadores, mas outra parte - talvez a mais importante - se deve a aplicação de técnicas para exploração das múltiplas unidades de computação para solução de um mesmo problema através do paralelismo. Um paradigma bastante estudado para modelagem de problemas que explora a localidade e evolução paralela de soluções é o dos autômatos celulares [Cannataro et al. 1995].

A. Autômatos Celulares

Autômatos celulares são modelos de computação discretos, concebidos por John von Neumann e Stanislaw Ulam [5]. Por possibilitarem a modelagem de sistemas com interações complexas e não-lineares, foi utilizado para aplicações em diversas áreas como física [7], biologia [6] e geografia [4].

Em sua definição formal, um autômato celular consiste em uma grade regular de células, que podem assumir um número finito de estados. O tempo é discreto e evolui em gerações. Cada célula calcula seu estado na próxima geração de acordo com sua vizinhança e uma regra de evolução. Na prática, essas características são flexíveis e dependem da aplicação. Diferentes implementações podem permitir, por exemplo, um número infinito de estados representados por um valor contínuo [9] ou até evolução assíncrona das células [1]. Neste trabalho, é utilizada a definição padrão dos autômatos celulares e o conjunto de regras utilizado é conhecido como o Jogo da Vida.

B. Jogo da Vida

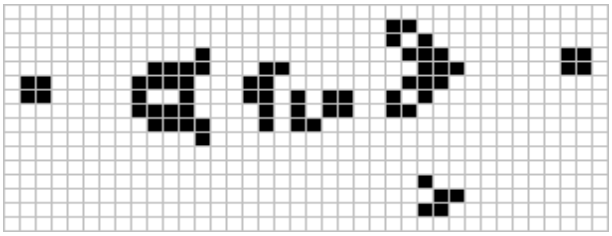

Fig. 1. Padrão "Glider Gun", descoberto por Bill Gosper em 1970 [10]. Este foi o primeiro padrão finito de crescimento infinito descoberto.

O Jogo da Vida é um conjunto de regras para autômatos celulares proposto por John Conway em 1970 [2]. Em sua implementação é utilizada a vizinhança de Moore [8] e as regras que definem o estado de uma célula na próxima geração se resumem a:

(i) Células vivas com 2 ou 3 vizinhos vivos continuam vivas;
(ii) Células mortas com exatamente 3 vizinhos vivos vivem;
(iii) Células com quaisquer outras configurações morrem.

Este conjunto de regras define uma das mais conhecidas e estudadas aplicações de autômatos celulares [REF]. Portanto, seu uso como objeto de estudo pode oferecer observações úteis para a estimativa dos speedups e fatores de eficiência que são possíveis através da implementação de autômatos celulares em hardware paralelo.

C. Objetivo

O objetivo deste trabalho é explorar técnicas de paralelismo para desenvolver uma aplicação para simulação de autômatos celulares em CPUs multi-core e GPUs CUDA. Em específico, o autômato celular estudado será o Jogo da Vida. Serão analisados os tempos de execução, speedups e fatores de eficiência, considerando diferentes tamanhos de grade raios de vizinhança e processos relacionados à renderização. Esses testes buscam, respectivamente, entender como o desempenho é impactado pela variação na dimensão dos dados, gargalos de acesso à memória e sobreposição de tarefas.

II. Trabalhos relacionados

Autômatos celulares (CA) têm sido amplamente utilizados para modelar vários fenômenos nas ciências naturais, como dinâmica de fluidos, dinâmica molecular e formação de padrões. Nos últimos anos, tem havido um interesse crescente em explorar o potencial de hardware paralelo, como GPUs e CPUs multicore, para acelerar simulações de CA.

Cannataro et al. [1] propuseram o CAMEL, um ambiente de software escalável para simulações de autômatos celulares implementado em um computador paralelo baseado em Transputer. Os autores mostraram que o CAMEL alcançou alta performance em hardware paralelo, permitindo que os usuários programem aplicações de ciência computacional de forma transparente.

Gibson et al. [2] investigaram a implementação eficiente de CA em CPUs multicore e GPUs. Os autores examinaram a escalabilidade de CA em relação a parâmetros padrão, como tamanho de grade e vizinhança, número de estados e gerações. Eles descobriram que as GPUs mostraram variação mínima no tempo de processamento sob variação de atividade e que os aumentos de velocidade foram proporcionais ao tamanho da atividade / vizinhança aritmética.

Topa e Mlocek [3] demonstraram como um modelo existente de fluxo de água poderia ser implementado em GPUs com o quadro de programação OpenCL. Os autores mostraram que algoritmos de autômatos celulares, que são inherentemente paralelos, podem alcançar alta eficiência em GPUs.

Hariri et al. [4] propuseram uma arquitetura paralela para autômatos celulares de aprendizado baseados em computação evolutiva (CLA-EC) em FPGA. Os autores mostraram que a arquitetura proposta poderia resolver problemas de otimização milhares de vezes mais rápido do que implementações sequenciais.

Millán et al. [5] apresentaram uma implementação de CA que se saiu bem em cinco arquiteturas NVIDIA GPU diferentes, de Tesla a Maxwell, simulando sistemas com até um bilhão de células. Os autores descobriram que o uso de memória compartilhada e algoritmos Multicell não melhorou o desempenho em Fermi ou arquiteturas mais recentes.

Neste artigo, propomos uma nova implementação de autômatos celulares para GPUs e CPUs multicore que explora os pontos fortes de ambas as arquiteturas. Construímos sobre o trabalho anterior examinando o desempenho de nossa implementação em relação a diferentes parâmetros de CA e comparando-a a implementações existentes.

III. Implementação

Duas soluções paralelas foram implementadas: a primeira utilizando OpenMP para execução em CPUs multi-core e a segunda utilizando CUDA para execução em GPUs. Também foi implementada uma versão serial do algoritmo para fins de comparação. As implementações foram desenvolvidas em C++ 17 e utilizam OpenGL para renderização. O código desenvolvido está disponível no Github[1] e também em um notebook do Google Colab[2].

O fluxo do programa final compartilha diversas estruturas entre as versões paralelas para CPU, GPU e a versão serial. A execução começa carregando as opções passadas por linha de comando, como tamanho da grade, número de gerações, probabilidade de inicialização, entre outros[3]. Em seguida, é realizada a inicialização das classes dos autômatos e de visualização, onde é alocada a memória que armazenará o estado das células e os buffers de vértices que serão utilizados para renderização. São alocados dois arrays 2D vetorizados do tipo unsigned char (1 byte), um para armazenar o estado atual do autômato - que será lido para gerar os próximos estados e para renderização - e outro array para armazenar o próximo estado de todas as células. Com a grade alocada, é feita a leitura do arquivo que definirá o padrão inicial do autômato a ser evoluído. O loop que executa os métodos para evolução e para renderização do autômato é então iniciado, e é executado até que seja alcançado o número máximo de gerações ou o usuário cancele a execução. Por fim, os resultados são impressos e toda a memória alocada é liberada. Para grades relativamente pequenas é possível imprimir o estado final do autômato no próprio terminal, com a opção *-o*.

Os algoritmos para evolução dos autômatos e para atualização dos buffers seguem estruturas semelhantes na versão serial e nas versões paralelas. O algoritmo de evolução itera sobre todas as células contando a quantidade de vizinhos vivos e aplicando as regras descritas na Subseção I-B. Células da extremidade da grade são sempre consideradas inativas. O algoritmo de atualização de buffers também itera sobre todas as células da grade e calcula para qual vértice seu estado deverá ser mapeado. O método então executa uma operação atômica que atualiza o estado do vértice calculando o máximo entre o estado atual e o estado da célula avaliada. A diferença na implementação desses algoritmos para as diferentes versões do código se resume a como é feita a iteração sobre os arrays, o que será discutido nas seções subsequentes.

A. Implementação com OpenMP

A implementação com OpenMP utiliza diretivas *#pragma* para paralelizar os loops que iteram sobre o array para a evolução do autômato e para atualização dos buffers de renderização. Para acompanhar a contagem de células ativas é utilizado a cláusula *reduction* para evitar o uso de seções críticas. O programa permite a inserção de ruído aleatório durante a evolução das células, que ativa células mortas com uma pequena probabilidade. Para isso foi utilizada a função de geração de números aleatória *rand_r*, que é thread-safe mas requer que o usuário mantenha um registro do estado aleatório de cada thread utilizada. Por fim, o escalonamento estático de threads foi utilizado uma vez que é esperado que todas as threads executem a mesma quantidade de operações.

[1] https://github.com/bryanoliveira/cellular-automata

[2] https://bit.ly/CellularAutomataColab

[3] A lista completa de opções do programa pode ser visualizada executando o programa com a opção *-h* ou *–help*.

B. Implementação com CUDA

A implementação com CUDA utiliza kernels lançados com configuração unidimensional para facilitar o cálculo do número de blocos necessários para execução. O número de threads por bloco (t) é fixo e é determinado experimentalmente (Subseção V-A). O número de blocos (b) é definido pela dimensão dos dados (N) e número de Stream Processors (SPs) disponíveis na placa de vídeo testada, como definido na Equação 1. O uso da ferramenta oficial para cálculo de configuração de lançamento de maior ocupação [3] foi considerado, mas testes preliminares indicaram desempenho pior comparado à heurística apresentada[4].

$$b = min(\frac{N + t - 1}{t}, SPs) \tag{1}$$

Os kernels utilizam stride para suportar autômatos de tamanhos variados sem modificar a geometria das configurações de lançamento. Cada thread é responsável por contar os vizinhos ativos de suas respectivas células e aplicar a regra do autômato. Para a inserção do ruído e aleatoriedade thread-safe foi utilizado a biblioteca *curand_kernel*. Para a contagem de células ativas foi utilizado a função *atomicAdd*. Kernels para evolução do autômato, atualização de buffers de renderização e inicialização do autômato utilizam device functions para facilitar o reuso do código. Os kernels para evolução do autômato e para atualização dos buffers rodam em diferentes streams.

C. Renderização

Para a renderização foi utilizada a biblioteca OpenGL com *buffer objects*. Os buffer objects permitem a definição prévia dos vértices das figuras a serem renderizadas sem que seja necessário gerar iterativamente todas as formas a cada frame. Para isso, foi necessário o desenvolvimento de shaders que permitem o controle da cor de cada vértice, representando o estado das células do autômato. Durante o loop principal é feita a cópia do estado da grade atual para o estado dos vértices. Na implementação em CPU esse processo pode ser acelerado com OpenMP, como foi descrito anteriormente. Para a implementação em CUDA foi utilizada a biblioteca CUDA-GL Interop, que permite a atualização dos buffers de renderização dentro da GPU sem a necessidade de cópias para a memória do host.

Outro detalhe importante sobre a renderização é a otimização do uso da memória de vídeo[5]. Nesta implementação, o número de vértices utilizados para renderização é definido pela resolução da janela de exibição invés do número de células na grade. Para mapear as células do autômato para cada vértice foi implementado um algoritmo de projeção semelhante a uma operação de *max-pooling* [11] da grade do autômato para a grade de vértices.

[4]Pela alta dimensão dos dados utilizados para teste, as sugestões da ferramenta ultrapassam em centenas de vezes a quantidade de núcleos físicos da placa utilizada. O overhead do gerenciamento dessa grande quantidade de blocos impacta negativamente a performance da aplicação. Por isso, o limite superior do número de blocos definido pelo número de SPs foi introduzido.

[5]Uma forma simples para a definição do número de vértices necessários para representar o autômato é simplesmente mapear cada célula a um vértice. No entanto, dessa forma a memória utilizada para renderização cresceria de forma proporcional ao tamanho da grade. Além disso, ao considerar que resoluções de telas comuns têm aproximadamente 1920x1080 pixeis, uma grande quantidade de vértices de uma grade 4096x4096 nem seria renderizada.

IV. Experimentos

Os experimentos avaliam o desempenho extra que pode ser obtido através das implementações paralelas. O primeiro experimento avalia a melhor configuração de lançamento dos kernels CUDA, que será utilizada para os experimentos subsequentes. A partir destes resultados, é avaliado o impacto do tamanho da grade e diferentes raios de vizinhança sobre o desempenho.

Para isso, é medido o speedup S, definido pela Equação 2, onde T_s é o tempo de execução serial (em CPU) e T_p é o tempo de execução paralela (em CPU multi-core e GPU).

$$S = \frac{T_s}{T_p} \tag{2}$$

Além do speedup também é medida a eficiência E, definida pela Equação 3, onde p é o número de processadores considerado. Na implementação em CPU o número de processadores considerado é definido pelo número de threads. Na implementação em GPU o número de processadores é definido pelo número de núcleos CUDA.

$$E = \frac{S}{p} \tag{3}$$

Cada avaliação de desempenho compara a implementação serial com a de GPU e a implementação de CPU com 2, 4, 8, 12 e 16 threads. Todos testes que variam o tamanho da grade são executados para grades de tamanho $(2^i)^2$ para $i \in \{5, 6, ..., 12\}$ (i.e. 32x32, 64x64, ..., 4096x4096). Cada teste foi repetido cinco vezes[6] com as mesmas configurações para suavizar possíveis ruídos do sistema e processos em segundo plano. Além disso, todas as execuções evoluem a grade por mil gerações para permitir a estabilização dos padrões do autômato. As grades são inicializadas aleatoriamente e a probabilidade de uma célula ser inicializada como viva é de 50%.

A estação utilizada nos experimentos possui uma CPU AMD Ryzen 7 3700X (2019) com 8 núcleos, 16 threads e 32 MB de cache L3. A CPU opera com clocks entre 3,6 GHz e 4,4 GHz e trabalha sobre 32 GB de RAM DDR4 a 3200 MHz. Para os testes em GPU foi utilizada uma placa NVIDIA RTX 3080 (2020) com 8704 núcleos CUDA e 128 KB de cache L1 para cada SM, que são 68 no total. A GPU opera com clocks entre 1,45 GHz a 1,71 GHz sobre 10 GB de VRAM GDDR6X a 1188 MHz.

V. Resultados

Os resultados são divididos entre testes de configurações de lançamento, tamanhos de grade, renderização e raios de vizinhança. Exceto pelos testes de renderização, todos os outros testes são executados sem renderização (headless).

[6]Nos gráficos, os pontos representam a média entre as cinco execuções e as bandas representam o desvio padrão entre elas.

A. *Configurações de lançamento*

O primeiro experimento executado avalia os tempos médios de evolução pelo número de threads por bloco. O objetivo deste experimento é definir a configuração de lançamento de kernels de GPU que a possibilite alcançar o melhor desempenho para as avaliações subsequentes. A Figura 2 combina os resultados dos testes executados para grades de tamanho definido na seção Experimentos por mil gerações.

As configurações com 128, 192 e 256 threads por bloco obtiveram os menores tempos médios. Dado que a configuração com 256 threads por bloco alcançou a menor média e com fim de maximizar a ocupação da GPU, esta foi selecionada para todos os testes subsequentes.

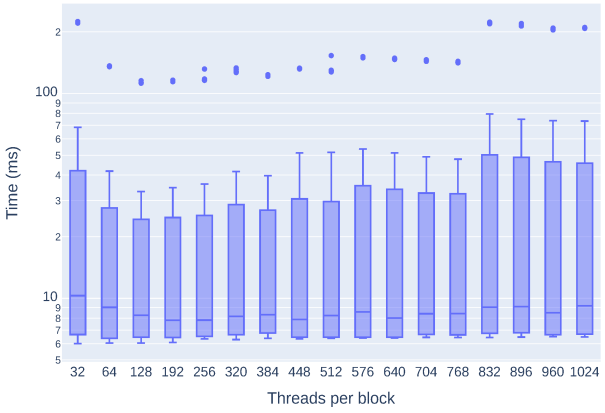

Fig. 2. Tempos médios de evolução por número de threads por bloco, considerando cinco execuções por mil gerações e todos os tamanhos de grade. O eixo Y (tempo) se encontra em escala logarítmica.

B. *Tamanhos de grade*

Foram avaliados os tempos médios por geração por tamanho de grade para as implementações em CPU e GPU. O objetivo deste experimento é entender como a dimensão dos dados afeta a performance das implementações paralelas e serial. A Figura 3 ilustra o comportamento das diferentes implementações a medida que a dimensão dos dados processados aumenta.

Um ponto a se notar é como os tempos para as implementações com mais threads (e.g. 16 CPUs e GPU) começa a subir de forma notável somente após um crescimento significativo dos dados. Outro aspecto relevante desse resultado é o tempo adicional para processamento de grades pequenas por essas mesmas implementações. Isso indica que overhead para gerenciamento de muitas threads pode não valer a pena para dimensões pequenas.

Uma outra forma de comparar os tempos de evolução é através do speedup obtido por cada implementação multi-thread sobre a implementação com 1 CPU (serial), como mostrado na Figura 4. É possível notar um ganho expressivo de desempenho de até 627 vezes da implementação em GPU sobre a implementação serial. As implementações em CPU multi-core também demonstraram ganhos de até 7.5 vezes com as configurações de 8 e 16 threads.

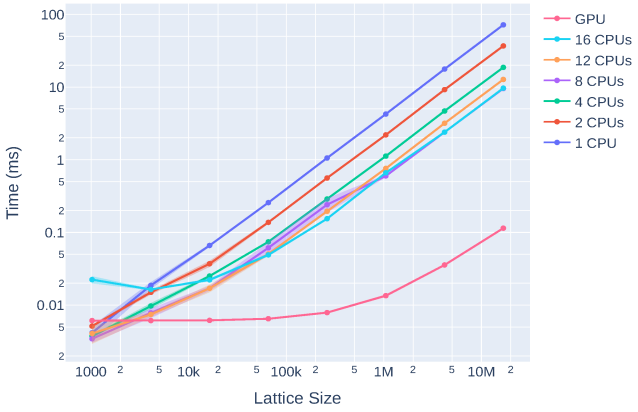

Fig. 3. Tempos médios por geração por tamanho de grade para implementações em CPU multi-core e GPU. Os eixos X (tamanho da grade) e Y (tempo) estão em escala logarítmica.

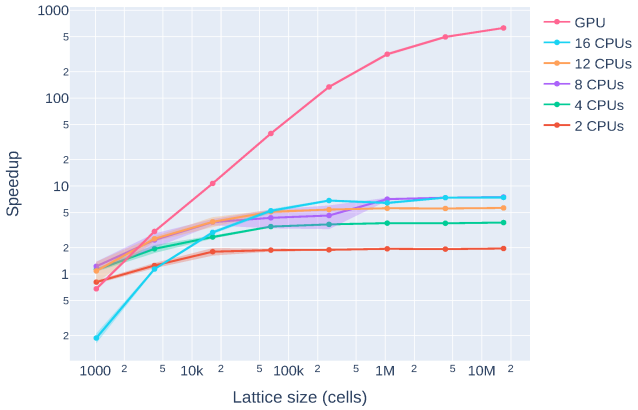

Fig. 4. Speedup sobre a implementação serial por tamanho de grade para implementações em CPU multi-core e GPU. Os eixos X (tamanho da grade) e Y (speedup) estão em escala logarítmica.

C. *Renderização*

Os experimentos com renderização investigam os ganhos na paralelização das cópias de buffers de vídeo e como a sobreposição de tarefas pela GPU pode afetar seu desempenho. A sobreposição de tarefas é mencionada pois, na estação utilizada para os experimentos, é papel exclusivo da GPU processar a visualização da grade e exibi-la em tela. Esse processo acontece tanto na implementação em CPU como na implementação em GPU. A Figura 5 compara o speedup das diferentes implementações sobre a implementação serial no processo de evolução das células do autômato.

Este resultado indica um impacto negativo no speedup da implementação em GPU para a evolução das gerações, enquanto o speedup das implementações em CPU se mantém semelhante aos testes headless. Ainda assim, a paralelização em GPU se mantém interessante para grades relativamente grandes, obtendo um speedup máximo de 415 vezes sobre a implementação serial.

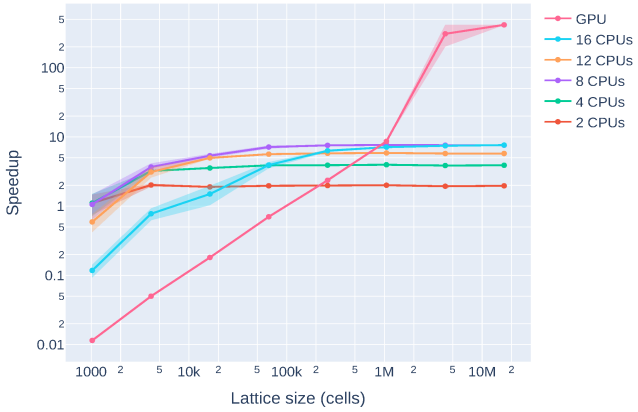

Fig. 5. Speedup sobre a implementação serial por tamanho da grade para implementações em CPU multi-core e GPU, com renderização habilitada. Os eixos X (tamanho da grade) e Y (speedup) estão em escala logarítmica.

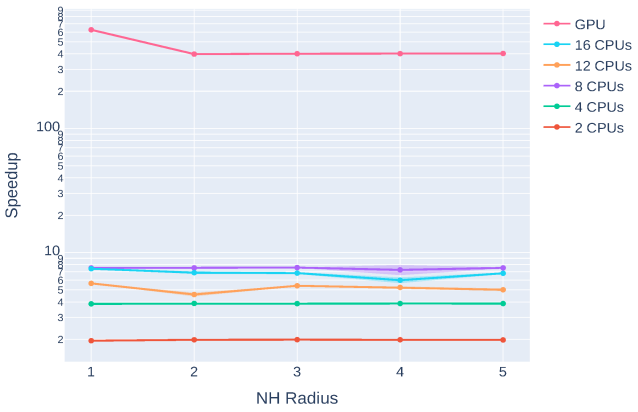

Fig. 7. Speedup sobre a implementação serial por raio de vizinhança para implementações em CPU multi-core e GPU. O eixo Y (tempo) está em escala logarítmica.

D. Raios de vizinhança

Os experimentos sobre diferentes raios de vizinhança avaliam o impacto do acesso à memória global pelas threads. O número mínimo de células lidas por cada valor de raio de vizinhança é descrito pela Tabela I.

A Figura 6 apresenta o tempo de processamento das diferentes implementações pelo raio de vizinhança testado, enquanto a Figura 7 apresenta seus respectivos speedups sobre a implementação serial. Todos os testes foram executados para um grade de tamanho 4096x4096.

TABELA I
NÚMERO DE CÉLULAS LIDAS POR RAIO DE VIZINHANÇA.

Raio	Células lidas
1	9
2	25
3	49
4	81
5	121

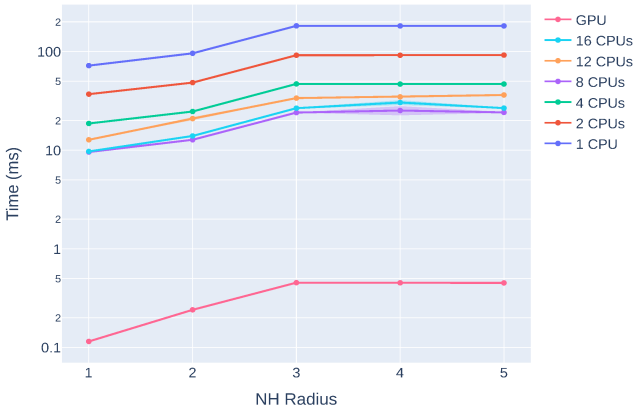

Fig. 6. Tempos médios por geração por raio de vizinhança para implementações em CPU multi-core e GPU e tamanho de grade 4096x4096. O eixo Y (tempo) está em escala logarítmica.

O aumento no número de leituras impacta negativamente o desempenho de forma semelhante em todas as implementações, mas não tanto quanto o esperado. Uma possível explicação para isso é o eficiente uso de cache pelo sistema, tanto por parte da CPU como pela GPU. Além disso, o padrão de acesso à memória favorece a localidade dos dados uma vez que threads vizinhas em mapeamento acessam células vizinhas da grade.

VI. Conclusões

Os resultados apresentados neste trabalho demonstram a possibilidade de ganhos significativos de desempenho na execução de autômatos celulares ao explorar o paralelismo de hardwares atuais. Speedups de até 627 vezes em GPU e até 7,5 vezes em CPU multi-core foram alcançados mesmo sem a implementação de controles avançados para o hardware utilizado.

Em relação à renderização, os resultados mostram vantagem para o uso da CPU multi-core quando a grade é relativamente pequena. Isso provavelmente se deve ao fato do processo de renderização sempre acontecer na própria GPU, forçando-a a sobrepor as tarefas de evolução e renderização do autômato. Desse modo, o tamanho da grade ideal para execução em CPU ou GPU depende de suas respectivas especificações e deve ser definido através de testes para cada aplicação.

Os experimentos com os diferentes raios de vizinhança demonstraram que o acesso à memória global para leitura de vizinhos não apresenta um gargalo considerável, o que foi inesperado. Isso possivelmente se deve a um eficiente gerenciamento de cache pelo sistema e pelo padrão de acesso à memória, em que threads vizinhas em mapeamento acessam células de memória vizinhas.

VII. Trabalhos futuros

Diversas direções para extensões do programa e trabalhos futuros surgem a partir desses resultados e das decisões de projeto tomadas. Uma otimização importante a ser considerada

é o uso de estruturas de dados mais econômicas para armazenamento do estado das células. Para autômatos com dois estados, como o Jogo da Vida, os estados das células podem ser armazenadas em um único bit, o que reduziria o consumo de memória em relação à implementação apresentada em até 8 vezes. O uso da memória compartilhada da GPU também pode ser explorado assim como a otimização dos padrões de acesso à memória pelas células. Por fim, pode ser avaliado o uso da memória de texturas da GPU para armazenamento das grades para a contagem de vizinhos. Isso pode ser interessante pois, pelo fato da GPU ser um hardware dedicado para tal, a aplicação de filtros que façam a contagem de vizinhos durante a leitura da grade pode ser mais eficiente do que os acessos sequenciais da implementação apresentada.

AGRADECIMENTOS

Agradeço ao professor Dr. Wellington Santos Martins pela orientação e excelentes aulas que possibilitaram a realização deste trabalho e que abriram novas portas para meu crescimento profissional e acadêmico.

REFERÊNCIAS

[1] N. Fatès, "Asynchronous cellular automata". *Encyclopedia of Complexity and Systems Science*, Springer, pp.21, 2018, 978-3-642-27737-5. 2017.

[2] M. Gardner, "The fantastic combinations of John Conway's new solitaire game 'life'". *Mathematical Games - Scientific American*, Vol. 223 no. 4. pp. 120–123. 1970.

[3] M. Harris, "CUDA Pro Tip: Occupancy API Simplifies Launch Configuration". *NVidia Developer Blog*. 2014. Disponível em https://developer.nvidia.com/blog/cuda-pro-tip-occupancy-api-simplifies-launch-configuration/. Acessado em jul/2021.

[4] R. Koenig, M. Daniela, "Cellular-automata-based simulation of the settlement development in Vienna". *Cellular Automata - Simplicity Behind Complexity*, A. Salcido (Ed.), pp. 23–46. 2011.

[5] J. V. Neumann, A. W. Burks, "Theory of Self-Reproducing Automata". *University of Illinois Press, USA*. 1966.

[6] E. Roberts, J.E. Stone, L. Sepúlveda, W.-M.W. Hwu, Z. Luthey-Schulten, "Long time-scale simulations of in vivo diffusion using GPU hardware". *Parallel & Distributed Processing*. IPDPS 2009. IEEE International Symposium on Rome. 2009.

[7] S. Szkoda, Z. Koza, M. Tykierko, "Accelerating cellular automata simulations using AVX and CUDA". *Arxiv*. 2012.

[8] E. W. Weisstein, "Moore Neighborhood". *MathWorld - A Wolfram Web Resource*. Disponível em https://mathworld.wolfram.com/MooreNeighborhood.html. Acessado em jul/2021.

[9] S. Wolfram, "A new kind of science". *Wolfram Media Inc.*, Champaign, Ilinois, USA. 2002.

[10] "Gosper glider gun". *LifeWiki*. 2020. Disponível em https://conwaylife.com/wiki/Gosper_glider_gun. Acessado em jul/2021.

[11] "Max-pooling / Pooling". *Computer Science Wiki*. 2018. Disponível em https://computersciencewiki.org/index.php/Max-pooling_/_Pooling. Acessado em jul/2021.