UNIVERSIDADE FEDERAL DE GOIÁS
INSTITUTO DE INFORMÁTICA - INF
PROGRAMA DE PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO
ALUNO: CARLOS HENRIQUE RORATO SOUZA **MATRÍCULA:** 2022100759

ComputerGO - Implementação Paralela em CUDA[1]

Resumo: *O problema ComputerGo, que consiste na construção de algoritmos que decidem jogadas para o jogo de tabuleiro chinês Go, é um problema que vem sido explorado na Computação sob diversas abordagens, na busca por algoritmos com bom desempenho (tempo de execução) e com boas respostas, que conduzam o jogador-máquina à vitória, ainda que esteja competindo com bons jogadores da "vida real". O presente artigo visa apresentar uma abordagem baseada em árvores de decisão, percorridas utilizando-se técnicas de Computação Paralela (linguagem CUDA) e o algoritmo Monte Carlo Tree Search. Um experimento foi conduzido realizando-se testes com quantidades diferentes de simulações (de 10 a 1.000.000 por nó). Os resultados apontam para uma redução de tempo que varia entre 4 e 5 vezes em relação ao algoritmo sequencial equivalente.*

Palavras-chave: *Computação Paralela, ComputerGO, Monte Carlo Tree Search, CUDA.*

1. INTRODUÇÃO:

Um tabuleiro com 19 x 19 intersecções, pedras brancas e pretas: eis o necessário para uma partida do jogo chinês Go. Também chamado de Weiqi ou Baduk, o jogo de estratégia e conquista territorial tem sua origem há cerca de 2 mil e 500 anos [1]. Ele não possui movimentos: trata-se apenas de posicionar as peças brancas e pretas nas intersecções, com o objetivo de cercar e capturar conjuntos de peças do adversário, e isso pode acontecer de diversas formas. A Figura 1 ilustra o tabuleiro com suas peças durante uma partida do jogo.

É um jogo de controle de território do tabuleiro: as pedras da mesma cor que são dispostas lado a lado vão formando blocos, que devem ir fechando os espaços chamados de "liberdades" (intersecções que ainda não foram preenchidas com pedras). Quando um dos jogadores ocupa todas as liberdades de uma pedra ou bloco, eles são capturados pelo oponente e retirados do tabuleiro. O objetivo é controlar a maior porção do território. O jogo termina quando não há mais movimentos a serem feitos pelos jogadores, isto é, quando começa um processo de "passar a vez" consecutivo.

[1] Projeto desenvolvido como trabalho final da disciplina de Computação Paralela (PPGCC-INF/UFG), ministrada pelo Prof. Dr. Wellington Santos Martins em 2022-1.

A posição onde são colocadas as pedras é a grande variável que define uma boa estratégia para vencer o jogo, tanto para proteger os seus territórios quanto para facilitar a captura dos territórios do oponente [1].

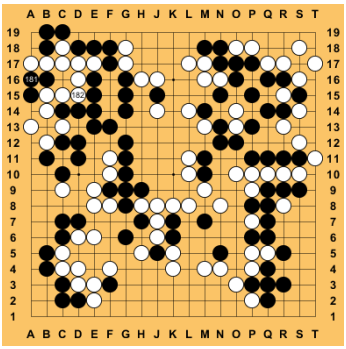

Figura 1 - Tabuleiro de Go [2].

Em termos computacionais, este jogo se tornou alvo de diversas abordagens envolvendo algoritmos e é ainda tópico de pesquisa por diversas questões, dentre as quais os autores elencam três: o espaço de movimentação (tabuleiro) é demasiadamente extenso, resultando numa quantidade alta de possibilidades de movimentação (o número de jogos possíveis gira em torno de 10^{171}) e isso o classifica como um problema PSPACE-difícil [1]. Além disso (e por conta disso também), se torna difícil a concepção de estratégias a longo prazo, e determinar o fim do jogo da forma tradicional (com os jogadores “passando a vez”) também é contra-intuitivo para um algoritmo, dado que não existe um “estado de jogo” que indica o término (como o Xadrez teria um cheque-mate, por exemplo). Segundo Johnson et al., o problema “Computer Go” continua em aberto [1].

A maior parte das soluções propostas utilizam as Game Trees, um tipo de árvore de decisão que armazena todos os possíveis estados de jogo. As folhas são os estados finais do jogo (vitórias e derrotas com os possíveis scores) e o desafio é: dado que se está em algum dos estados da árvore (estado de jogo atual), para que se conheça os caminhos possíveis que levam à vitória, é preciso percorrer todas as subárvores deste estado e, assim, o jogador pode escolher o melhor movimento a ser feito, que conduzirá ao maior *score*.

Nesse horizonte, este artigo apresenta uma abordagem para o problema ComputerGO utilizando conceitos de Computação Paralela na GPU (realizada em CUDA), a partir do método de busca em árvores Monte Carlo Tree Search (MCTS). A ideia principal consiste em realizar o percurso na árvore e as simulações, que são típicas deste método de busca, mapeando um nó da árvore para cada bloco disponível

na GPU, realizando as simulações em paralelo utilizando threads. A solução aqui descrita foi inspirada pelo trabalho de Johnson et al. [1], que utilizou o algoritmo Alfa-Beta-Pruning, também implementado em CUDA. Muitas das decisões de projeto tiveram como base a sua abordagem, inclusive em relação às limitações de memória na representação da árvore.

Para verificar o desempenho da solução desenvolvida, foi realizado um experimento a partir da execução do algoritmo com números de simulações por nó variando de 10 a 1.000.000. O tempo gasto foi comparado com uma versão sequencial do algoritmo implementada utilizando-se também o MCTS. Ao final, foi possível determinar o *speedup* e verificar os ganhos com a versão paralela.

Para a implementação, considerou-se uma versão simplificada do jogo: é considerada a captura de somente uma peça por vez (que precisa estar cercada na horizontal e vertical pelas peças do oponente) e o fim de jogo é determinado pela quantidade de jogadas (no máximo N x N jogadas, onde N é a dimensão do tabuleiro, neste caso 19), a exemplo de Johnson et al. [1]. Acredita-se que estas simplificações não impactam na relevância da solução desenvolvida, e considera-se que os ganhos obtidos possam ser refletidos de alguma forma em uma instância mais “completa” do jogo, isto é, com todo o conjunto de regras.

Este artigo está organizado em outras cinco seções. A Seção 2 discute o algoritmo proposto, a partir do método Monte Carlo Tree Search e das técnicas de Computação Paralela. A Seção 3, por sua vez, apresenta o experimento realizado, cujos resultados são discutidos na Seção 4. As considerações finais e referências são apresentadas nas seções 5 e 6, respectivamente.

2. ALGORITMO PROPOSTO: MCTS PARALELO

Para a implementação da busca na árvore de decisão, optou-se pelo algoritmo Monte Carlo Tree Search (MCTS). Ele tem sido vastamente utilizado em problemas modelados a partir da pesquisa em árvores com adversários, tendo ganhado um grande destaque pelos bons resultados obtidos em jogos [3], como o Go, tendo sido utilizado no AlphaGo[2], um dos softwares mais conhecidos e com um melhor desempenho para este jogo. Sobre a versatilidade do algoritmo, Moreira coloca que:

> O MCTS pode ficar a correr indefinidamente até encontrar uma resposta ótima, ou podemos limitar o tempo ou o número de iterações do algoritmo, fazendo com que este retorne sempre uma resposta. Contudo, quanto mais tempo dermos ao algoritmo,

[2] https://www.deepmind.com/research/highlighted-research/alphago

> maior a probabilidade da resposta ser ótima, ou seja, se o tempo de execução tender para o infinito, o algoritmo irá retornar a melhor solução possível [4].

O algoritmo é simples, sendo composto de quatro passos principais. O Algoritmo 1 apresenta um pseudocódigo do MCTS, a partir destes quatro passos, adaptado de Moreira [4].

Algoritmo 1: MCTS (pseudocódigo)

```
Função MCTS(Estado S0):
  Cria o nó raiz V0 a partir de S0
  Enquanto Dentro dos limites estipulados faça
      V1 = PolíticaDeArvore(V0)
      D = PolíticaDeSimulação(Simular(V1))
      Propagar(D,V1)
  Retorna MelhorFilho(V0)
```

De maneira geral, são considerados os seguintes passos [3]:

- **Seleção:** a partir de um estado inicial, o algoritmo encontrará um nó expansível (deve ser uma folha da árvore atual cujos filhos ainda não foram visitados). Na implementação proposta por este trabalho, o nó selecionado é sempre o estado atual de jogo, no qual o jogador-pessoa acabou de fazer a sua jogada (tendo jogado uma peça preta) e o jogador-máquina precisa decidir a jogada (onde colocar uma peça branca). Assim, não é necessário percorrer a árvore para selecionar um elemento. A seleção é ilustrada na Figura 2 (a), e o nó selecionado está na cor laranja.
- **Expansão:** encontrado o nó expansível, uma quantidade predeterminada de filhos será gerada. Este é um ponto importante para uma boa resposta do algoritmo: quanto mais filhos forem expandidos, mais acertada será a resposta. Entretanto, isso impacta diretamente no desempenho do algoritmo. Para isso, é necessário existir uma **política de árvore**, que decide qual filho será expandido. Neste trabalho, considera-se como política de árvore a expansão de todos os filhos. Isso quer dizer que, no pior caso (tabuleiro vazio), serão gerados N*N filhos (N*N é a dimensão do tabuleiro), que são as N*N possibilidades de se colocar uma peça branca. Na Figura 2 (b), os nós expandidos estão coloridos de verde.

- **Simulação:** após a expansão dos nós, utiliza-se uma **política de simulação** para que sejam simulados K jogos completos (até o fim de jogo), a partir de cada nó expandido. Esse processo visa buscar a utilidade de cada um desses nós, para que seja possível uma comparação. Novamente, é um processo crucial para uma boa resposta por parte do algoritmo: quanto mais simulações (valor K), mais acertada será a resposta. Porém, o impacto no desempenho é alto, assim como o passo anterior. No pior caso, são K*N*N simulações. Neste trabalho, a política de simulação é a geração aleatória das jogadas, fazendo com que o algoritmo seja flexível para ser testado com valores diferentes de K e diminuindo a chance de que, dadas duas simulações, elas sejam idênticas. A média dos *scores* obtidos nas simulações (para cada nó) é armazenada para a próxima etapa. A Figura 2 (c) ilustra as simulações com a linha tracejada.
- **Propagação:** trata-se de propagar pela árvore os resultados da simulação, fazendo-se a escolha pelo melhor dos filhos, isto é, o que alcançou a utilidade mais adequada. No caso deste trabalho, a estrutura de dados que contém os *scores* médios de cada nó é percorrida e verifica-se qual nó possui a melhor pontuação. Este nó é escolhido e retornado pelo algoritmo como aquele que contém a jogada que deverá ser realizada. Na Figura 2 (d), o nó escolhido está em azul.

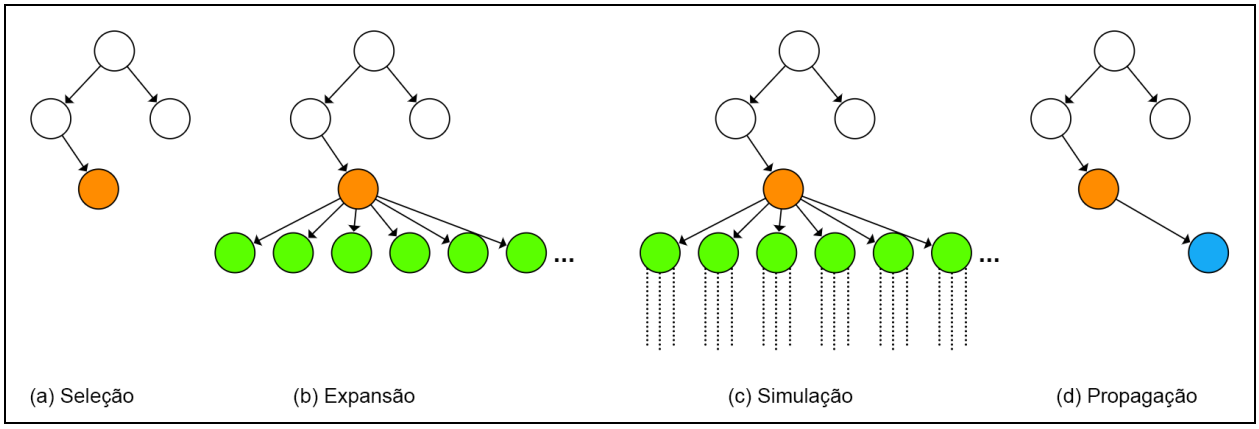

Figura 2 - Etapas do MCTS.

A implementação paralela foi realizada na linguagem CUDA, da NVIDIA[3], no contexto de paralelismo na GPU. Cada nó da árvore é representado por uma struct, que armazena um tabuleiro, a peça jogada e as coordenadas onde foi colocada, o nível

[3] https://developer.nvidia.com/cuda-zone

da árvore no qual esta configuração de jogo se situa e o *score* (subtração da quantidade de peças brancas e pretas) deste tabuleiro. A ideia principal é realizar as expansões e simulações de forma paralela, de maneira que cada nó expandido seja processado por um bloco e que as K simulações sejam divididas entre as *thread*s de cada bloco, e feitas em passadas (strides).

Um dos principais problemas do ComputerGO é a questão de armazenamento dos dados. Depois de diversas tentativas e problemas enfrentados, optou-se por seguir a lógica de Johnson et al. [1]: armazenar somente o estado atual de jogo, que é passado como parâmetro para a função MCTS, que o utiliza para realizar a geração temporária de todos os filhos e as simulações. O importante, neste momento, não é guardar cada estado correspondente ao filho, mas sim o score médio das simulações relativas a ele, além das coordenadas da nova jogada (de peça branca) que resultou neste filho.

Para realizar este processo, considerou-se a seguinte lógica: dado que são utilizados N*N blocos (que é a dimensão do tabuleiro), a jogada realizada pelo bloco de índice i é aquela na linha i/N e coluna i%N. Se essa posição já estiver ocupada, o bloco não será utilizado, e seu *score* recebe -666, marcando-o para não ser escolhido na etapa de propagação.

O *array* de *scores*, por sua vez, utiliza o recurso de Memória Unificada, pois é acessado tanto na CPU quanto na GPU. Ele possui tamanho igual à quantidade total de *threads* utilizadas, para que cada thread utilize a sua própria posição para guardar a soma das simulações que fez e não haja conflitos no acesso e armazenamento dos valores. Na fase de propagação, calcula-se a média de todas as simulações feitas em cada bloco, percorrendo-se este *array*.

Por questões de praticidade e conveniência, a fase de propagação do algoritmo foi “desmembrada” dos outros passos do MCTS, acontecendo na CPU de maneira serial, depois que os passos anteriores já foram concluídos na GPU. Ao final dessa fase, tem-se o índice i do bloco que teve a melhor pontuação. Ele foi o nó selecionado, e a jogada será realizada no tabuleiro com o estado atual, sendo adicionada a nova peça na linha i/N e coluna i%N. Parte-se, então, para a próxima jogada.

Por fim, é válido registrar que, antes da implementação paralela, uma versão sequencial deste algoritmo foi desenvolvida, visando o estudo e aprofundamento dos conceitos relativos ao MCTS e, ademais, para auxiliar na prevenção de erros de implementação da versão paralela e nos experimentos documentados neste trabalho.

3. EXPERIMENTO

Finalizada a etapa de implementação, iniciou-se a realização de um experimento com o objetivo de analisar o algoritmo desenvolvido e, comparando-o com uma versão não-paralela, verificar possíveis ganhos de desempenho.

O experimento foi realizado no ambiente do Google Colab[4], utilizando-se de uma máquina virtual cuja GPU de modelo Tesla T4 possuía as características especificadas na Figura 3 (resultado de execução do comando “nvidia-smi”). A versão da linguagem CUDA utilizada foi a 11.2.

```
Wed Jul 27 17:30:15 2022
+-----------------------------------------------------------------------------+
| NVIDIA-SMI 460.32.03    Driver Version: 460.32.03    CUDA Version: 11.2     |
|-------------------------------+----------------------+----------------------+
| GPU  Name        Persistence-M| Bus-Id        Disp.A | Volatile Uncorr. ECC |
| Fan  Temp  Perf  Pwr:Usage/Cap|         Memory-Usage | GPU-Util  Compute M. |
|                               |                      |               MIG M. |
|===============================+======================+======================|
|   0  Tesla T4            Off  | 00000000:00:04.0 Off |                    0 |
| N/A   53C    P8    10W /  70W |      0MiB / 15109MiB |      0%      Default |
|                               |                      |                  N/A |
+-------------------------------+----------------------+----------------------+
```

Figura 3 - Configuração da GPU utilizada no experimento.

Para a realização do experimento, foram consideradas 5 execuções do algoritmo sequencial (ES1,...,ES5) e 5 do algoritmo paralelo (EP1,...,EP5), variando-se a quantidade de simulações para cada nó (de 10 simulações até 1.000.000). Fez-se a média das execuções (sequenciais e paralelas), para que fosse possível calcular o speedup, que foi considerado como a razão entre o tempo médio sequencial e o tempo médio paralelo.

4. RESULTADOS E DISCUSSÃO

Em relação às execuções da versão sequencial do algoritmo, os tempos obtidos (em segundos) estão dispostos na Tabela 1. Observa-se que a proporção de aumento do número de simulações reflete-se também (de maneira aproximada) no tempo resultante, que varia desde 1,8 segundos (para 10 simulações por nó) até aproximadamente 5 horas para 100.000 simulações por nó e 47 horas para 1.000.000.

Tabela 1 - Resultado das execuções sequenciais (em segundos).

	ES1	ES2	ES3	ES4	ES5	Média (S)
10	1,883	1,905	1,902	1,869	1,892	1,890

[4] https://colab.research.google.com/

100	18,887	18,428	18,154	18,322	18,441	18,446
1000	180,889	180,984	179,997	180,008	180,445	180,465
10.000	1705,680	1706,440	1705,970	1705,740	1706,020	1705,970
100.000	17100,722	17100,849	17101,002	17100,437	17101,021	17100,806
1.000.000	171171,855	171172,004	171171,932	171173,112	171170,766	171171,934

Já em relação às execuções paralelas, observou-se uma redução considerável no tempo de execução. Os tempos de cada execução, a média (todos em segundos) e o speedup calculado estão dispostos na Tabela 2. Observou-se que, no caso de 10 simulações, a queda no tempo foi pequena (aproximadamente 42% de ganho), quando comparada aos ganhos de tempo com um número maior de simulações: as quase 5 horas do algoritmo sequencial (para 100.000 simulações) deram lugar a um tempo de aproximadamente 1 hora. Para 1.000.000 de simulações por nó, o ganho foi similar, saindo de 47 horas de execução para aproximadamente 9 horas e meia. A proporcionalidade entre o aumento do número de simulações e o crescimento do tempo também foi notada nas execuções paralelas.

Tabela 2 - Resultado das execuções paralelas (em segundos) e cálculo do *speedup*.

	EP1	**EP2**	**EP3**	**EP4**	**EP5**	**Média (P)**	**Speedup (S/P)**
10	1,319	1,330	1,320	1,341	1,314	1,325	1,427
100	4,022	4,076	4,043	4,079	4,021	4,048	4,557
1000	34,956	35,083	34,930	34,978	35,044	34,998	5,156
10.000	345,972	349,131	348,170	347,224	345,881	347,276	4,912
100.000	3499,681	3498,632	3500,235	3498,987	3500,012	3499,509	4,887
1.000.000	34559,480	35007,341	34567,298	34558,221	34559,689	34650,406	4,940

Em relação ao *speedup*, além da coluna na Tabela 2, um gráfico mostrando a sua variação de acordo com o número de simulações é mostrado na Figura 4, na qual o eixo X apresenta o número de simulações por nó e o eixo Y, por sua vez, o valor do *speedup*.

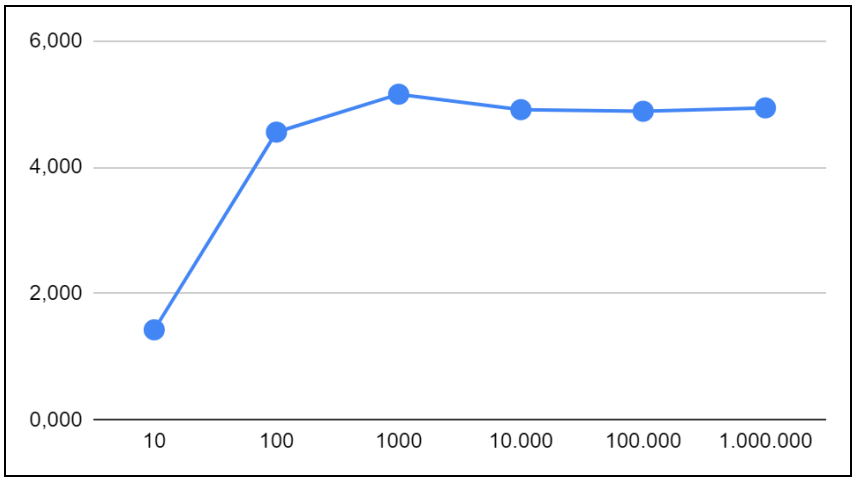

Figura 4 - Variação do speedup no experimento.

É possível notar um crescimento do speedup nos três primeiros valores (10,100 e 1.000 simulações). A partir deste momento, nota-se a sua estabilidade, alcançada nas proximidades de 5 (de 4,8 a 5,1), o que aponta para uma redução média de 5 vezes, em relação ao tempo de execução do algoritmo sequencial. Ademais, todos os valores obtidos caracterizam um *speedup* normal (1 < speedup < 19*19), isto é, o algoritmo proposto reduz o tempo de execução nos parâmetros da normalidade. Com isso, tem-se uma abordagem válida.

5. CONSIDERAÇÕES FINAIS

Este artigo apresentou uma abordagem para o problema ComputerGO, envolvendo o percurso na árvore de decisão do jogo a partir do método Monte Carlo Tree Search, implementado utilizando-se conceitos e técnicas de Computação Paralela. Após a realização de um experimento, foi possível verificar que o algoritmo apresenta ganhos consideráveis de desempenho, se comparado com uma versão sequencial do mesmo algoritmo.

A partir deste estudo, surgem diversas oportunidades de continuidade. Um primeiro passo seria expandir o conjunto de regras do jogo, ampliando-o e deixando-o mais próximo do GO “real”. Com um aprofundamento maior no método MCTS, seria possível otimizar as políticas de árvore e de simulação, bem como a realização de testes com um número maior de simulações, mais próximo da quantidade de jogos possíveis para cada subárvore. Tais refinamentos tornam mais favoráveis as condições para uma resposta melhor do algoritmo.

6. REFERÊNCIAS

[1] - Johnson, C., Barford, L., Dascalu, S.M., Harris, F.C. (2016). CUDA Implementation of Computer Go Game Tree Search. In: Latifi, S. (eds) Information Technology: New Generations. Advances in Intelligent Systems and Computing, vol 448. Springer, Cham.

[2] - Holzinger, A. (2016). January, 27, 2016, Major breakthrough in AI research. Disponível em: https://hum-an-centered.ai/2016/01/28/january-27-2016-major-breakthrough-in-ai-research/. Acesso em 24/06/2022.

[3] - Browne, C.B., Powley, E., Whitehouse D., et al. (2012). A survey of monte carlo tree search methods. Computational Intelligence and AI in Games, IEEE Transactions on, 4(1):1–43.

[4] - Moreira, J. M. C. (2016). Implementação e Avaliação do Algoritmo MCTS-UCT para o jogo Chinese Checkers. Dissertação de Mestrado: Departamento de Ciência de Computadores. Faculdade de Ciências da Universidade do Porto.