Game of Life: Implementação com OpenCL

José Alan Teixeira[1]

[1]Programa de Pós-graduação em Ciências da Computação

Universidade Estadual de Feira de Santana (UEFS)

Feira de Santana - BA - Brasil

{joseasteixeira7}@gmail.com

1. Introdução

O *Game of Life*, proposto por John Conway, é um autômato celular estudado em simulações de sistemas dinâmicos complexos. Sua simplicidade de regras e capacidade de gerar padrões complexos tornam-no ideal para explorar conceitos em computação paralela. Diante do crescimento da demanda por desempenho computacional, técnicas que exploram a capacidade massivamente paralela de GPUs vêm sendo cada vez mais aplicadas. Nesse contexto, o presente trabalho propõe uma implementação do *Game of Life* utilizando a plataforma OpenCL, que permite a execução paralela do código em dispositivos heterogêneos, como GPUs.

A proposta deste trabalho é descrever em detalhes a arquitetura do sistema implementado, abordando a implementação e funcionamento do software para o ambiente OpenCL. Além disso, são discutidas as estratégias de paralelismo adotadas, as vantagens do uso da GPU no contexto específico do *Game of Life*, bem como as instruções para execução e os resultados obtidos. O objetivo é fornecer uma visão clara e prática da utilização de OpenCL para simular sistemas computacionais dinâmicos, servindo tanto como base para trabalhos futuros quanto como material didático para o ensino de computação paralela.

2. Descrição do código

O código do openCL foi implementado com base na versão serial do código diponivel no link do Git Hub: https://github.com/joseasteixeira/GameOfLife.git.

2.1. Inicialização do Ambiente OpenCL

Função: inicializar_opencl(const char *nome_arquivo_kernel) configura o ambiente de execução OpenCL, selecionando a plataforma e o dispositivo, criando contexto, fila de comandos e construindo o kernel a partir do código fonte. A execução de código paralelizado com OpenCL requer a inicialização de uma GPU, além da compilação do kernel que será executado na GPU. Esse processo é necessário para preparar os recursos computacionais da GPU para uso.

Inicialmente a plataforma OpenCL é localiza e a GPU selecionada.

```
// Obter plataforma
err = clGetPlatformIDs(1, &platform, NULL);
if (err != CL_SUCCESS) { printf("Erro ao obter plataforma\n"); return 0; }

// Obter dispositivo
err = clGetDeviceIDs(platform, CL_DEVICE_TYPE_GPU, 1, &device, NULL);
if (err != CL_SUCCESS) { printf("Erro ao obter dispositivo\n"); return 0; }
```

Criação do contexto e da fila de comandos, contexto define o ambiente de execução, enquanto a fila é usada para enviar comandos de execução do kernel ou transferências de dados.

```
// Criar contexto
context = clCreateContext(NULL, 1, &device, NULL, NULL, &err);
if (!context) { printf("Erro ao criar contexto\n"); return 0; }

// Criar fila de comandos
queue = clCreateCommandQueueWithProperties(context, device, 0, &err);
if (!queue) { printf("Erro ao criar fila\n"); return 0; }
```

Leitura e compilação do kernel, o código do kernel é carrega do arquivo gol_kernel.cl, compila-o para o dispositivo e cria uma instância do kernel.

```
// Ler o código-fonte do kernel
FILE *f = fopen(gol_kernel, "r");
if (!f) { printf("Erro ao abrir kernel\n"); return 0; }
fseek(f, 0, SEEK_END);
size_t size = ftell(f);
rewind(f);
char *source = (char *)malloc(size + 1);
fread(source, 1, size, f);
source[size] = '\0';
fclose(f);

// Criar e compilar programa
program = clCreateProgramWithSource(context, 1, (const char **)&source, NULL, &err);
if (err != CL_SUCCESS) { printf("Erro ao criar programa\n"); return 0; }

err = clBuildProgram(program, 1, &device, NULL, NULL, NULL);
if (err != CL_SUCCESS) {
    // Imprimir log de erro
    char log[2048];
    clGetProgramBuildInfo(program, device, CL_PROGRAM_BUILD_LOG, sizeof(log), log, NULL);
    printf("Erro ao compilar kernel:\n%s\n", log);
    return 0;
}

// Criar kernel
kernel = clCreateKernel(program, "kernel_gol", &err);
if (!kernel || err != CL_SUCCESS) {
    printf("Erro ao criar kernel\n");
    return 0;
}
```

2.2. Função de Geração de Nova Iteração

Função: gen_next_gpu(Cell **grid, int height, int width) calcula a próxima geração do Jogo da Vida na GPU utilizando o kernel OpenCL, substituindo a grade atual pela nova geração. A execução do Jogo da Vida envolve cálculos independentes para cada célula.

Utilizar a GPU com OpenCL permite que as células sejam processadas simultaneamente, aumentando o desempenho.

Conversão da matriz 2D para vetor linear isso facilita a manipulação de dados pela GPU, que trabalha com memória linear.

```
// Converte a matriz de Cell para vetor plano de int
for (int i = 0; i < HEIGHT; i++) {
    for (int j = 0; j < WIDTH; j++) {
        grid_flat[i * WIDTH + j] = grid[i][j].state;
    }
}
```

Cria buffers na GPU, alocando a memória na GPU para armazenar a grade atual e a nova.

```
// Aloca memória na GPU
cl_mem d_grid = clCreateBuffer(context, CL_MEM_READ_ONLY |
    CL_MEM_COPY_HOST_PTR, size, grid_flat, &err);
checkError(err, "Criando buffer d_grid");

cl_mem d_new_grid = clCreateBuffer(context, CL_MEM_WRITE_ONLY, size, NULL, &err);
checkError(err, "Criando buffer d_new_grid");
```

Definição dos argumentos do kernel, especifica os dados que o kernel vai usar e buffers (tamanho da grade).

```
// Define os argumentos do kernel
err = clSetKernelArg(kernel, 0, sizeof(cl_mem), &d_grid);
err |= clSetKernelArg(kernel, 1, sizeof(cl_mem), &d_new_grid);
err |= clSetKernelArg(kernel, 2, sizeof(unsigned int), &WIDTH);
err |= clSetKernelArg(kernel, 3, sizeof(unsigned int), &HEIGHT);
checkError(err, "Definindo argumentos do kernel");
```

Definição do espaço de trabalho global define que cada célula da grade será processada por um thread independente.

```
// Define o tamanho global
size_t global_work_size[2] = {HEIGHT, WIDTH};
```

Enfileiramento da execução do kernel, envia o kernel para execução e espera ele terminar.

```
// Executa o kernel
err = clEnqueueNDRangeKernel(queue, kernel, 2, NULL, global_work_size,
    NULL, 0, NULL, NULL);
checkError(err, "Executando kernel");

clFinish(queue);
```

Copia o resultado da GPU para a CPU, recupera a nova geração da grade para uso no programa.

```
// Copia resultado de volta para a CPU
err = clEnqueueReadBuffer(queue, d_new_grid, CL_TRUE, 0, size, new_grid_flat,
    0, NULL, NULL);
checkError(err, "Lendo resultado do buffer");
```

Converte vetor linear para matriz 2D e reconstrói a representação bidimensional da grade.

```
// Atualiza a grade original
for (int i = 0; i < HEIGHT; i++) {
    for (int j = 0; j < WIDTH; j++) {
        grid[i][j].state = new_grid_flat[i * WIDTH + j];
    }
}
```

2.3. Código do Kernel OpenCL

Arquivo: gol_kernel.cl

Kernel: __kernel void kernel_gol(...) calcula o estado futuro de uma célula com base nos estados dos seus vizinhos. Cada thread calcula a próxima geração de uma única célula de forma independente.

```
1   __kernel void kernel_gol(
2       __global const int* grid_atual,
3       __global int* grid_prox,
4       const int width,
5       const int height
6   ) {
7       int x = get_global_id(0);
8       int y = get_global_id(1);
9
10      int vivos = 0;
11
12      for (int dx = -1; dx <= 1; dx++) {
13          for (int dy = -1; dy <= 1; dy++) {
14              if (dx == 0 && dy == 0) continue;
15
16              int nx = (x + dx + width) % width;
17              int ny = (y + dy + height) % height;
18
19              vivos += grid_atual[ny * width + nx];
20          }
21      }
22
23      int idx = y * width + x;
24      int estado = grid_atual[idx];
25
26      if (estado == 1) {
27          grid_prox[idx] = (vivos == 2 || vivos == 3) ? 1 : 0;
28      } else {
29          grid_prox[idx] = (vivos == 3) ? 1 : 0;
30      }
31  }
```

O kernel realiza as seguintes etapas:

- **Identifica a célula que a thread está processando**:

Usa get_global_id para obter as coordenadas (x, y) da célula atual.

- **Conta vizinhos vivos**:

Percorre as 8 vizinhanças ao redor da célula, aplicando condições de contorno periódicas com operadores módulos.

- **Aplica regras do Jogo da Vida**:

Com base no número de vizinhos vivos, determina se a célula vive, morre ou nasce.

- **Salva o resultado no buffer de saída**:

Atualiza a nova geração da grade no buffer correspondente.

4. Resultados

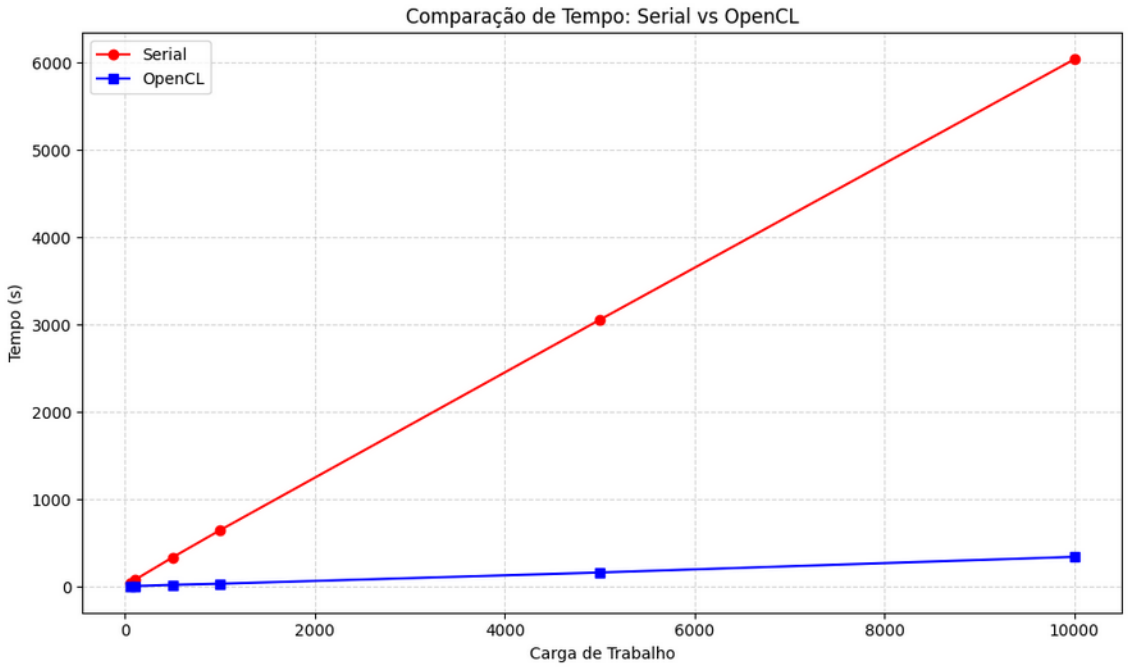
Para realizar os testes de execução a GPU utilizada possui as seguintes características:

- Name: Intel(R) UHD Graphics 620
- Version: OpenCL 3.0
- Max. Compute Units: 24
- Local Memory Size: 64 KB
- Global Memory Size: 6927 MB
- Max Alloc Size: 3463 MB
- Max Work-group Total Size: 256
- Max Work-group Dims: (256 256 256)

Para avaliar o desempenho da implementação com OpenCL em comparação com a versão serial do algoritmo do *Game of Life*, foram realizados experimentos com diferentes cargas. A tabela a seguir apresenta os tempos de execução (em segundos) observados para cada abordagem:

Tabela 1: Tempos de execução serial e com OpenCL para cada carga.

Carga	Tempo(s) Serial	Tempo(s) OpenCL
50	40,03	3,56
100	75,22	5,36
500	334,49	19,82
1000	645,80	32,34
5000	3056,41	161,01
10000	6041,14	340,43



Figira 1: Gráfico de comparação de desempenho.

Analisando a Figura 1 o gráfico mostra os ganhos de desempenho com o uso do OpenCL, especialmente para cargas maiores. Na carga de 5000 interações, por exemplo, a execução paralela reduziu o tempo de simulação em quase 95% em relação à versão serial. Mesmo em cargas menores, a aceleração com OpenCL já é perceptível, confirmando a eficiência da paralelização para esse tipo de problema.

Tabela 2: Speedup e Efeciência para cada carga.

Carga	Speedup	Eficiência
50	11.24	46.83
100	14.03	58.45
500	16.88	70.34
1000	19.97	83.22
5000	18.98	79.08
10000	17.74	73.92

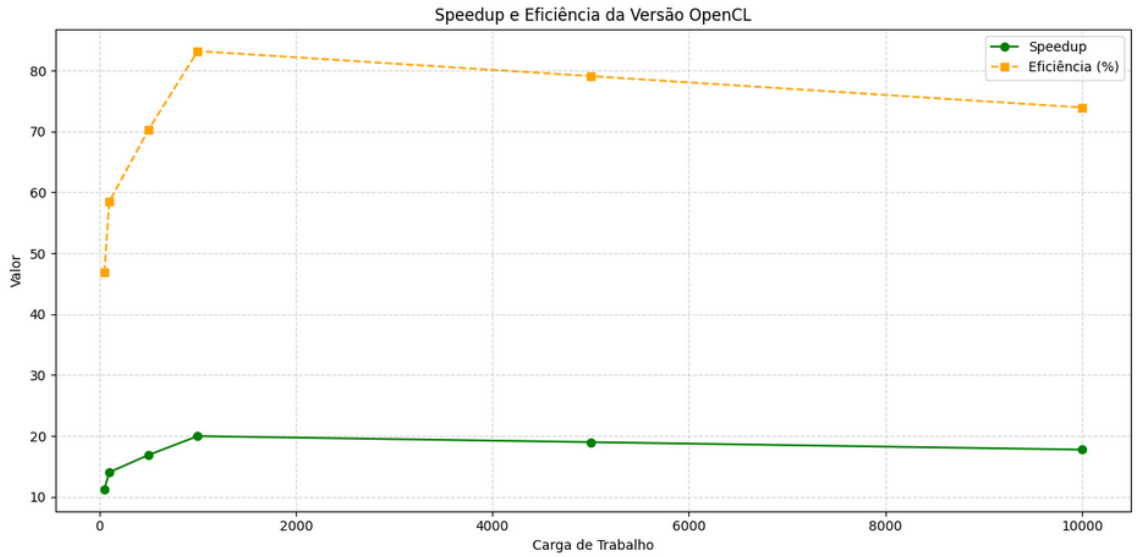


Figura 2: Gráficos do Speedup e da Eficiência

Observa-se que o speedup (razão entre o tempo serial e o tempo paralelo) aumentou conforme a carga de trabalho cresceu. Para uma carga de 50 iterações, o speedup foi de aproximadamente 11,24, enquanto para 10.000 iterações o valor alcançou 17,74. Esse comportamento acontece, pois, tarefas maiores tendem a aproveitar melhor os recursos de paralelismo, amortizando o overhead associado à inicialização e à comunicação entre as unidades de execução.

Além do speedup, também foi avaliada a eficiência paralela, definida como a razão entre o speedup obtido e o número total de unidades de execução disponíveis na GPU, que neste caso é de 24 unidades de computo. A eficiência variou de 46,83% para a carga 50 até cerca de 83,22% para a carga de 1000 iterações. Isso indica que, ouve um aproveitamento dos recursos com o crescimento da carga, até atingir um pico máximo e a eficiência começar a diminuir. Em resumo, o uso de OpenCL trouxe ganhos significativos de desempenho, principalmente para cargas maiores, a eficiência um bom aproveitamento da arquitetura da GPU mais que ainda poderia ser maiores ganho de desempenho.

5. Conclusão

A implementação do *Game of Life* com uso da tecnologia OpenCL demonstrou-se eficiente na aplicação de conceitos de paralelismo em GPUs, aproveitando seu potencial de processamento massivo para simulações de autômatos celulares. O desenvolvimento da aplicação contemplou todas as etapas essenciais para a execução paralela do algoritmo, desde a configuração do ambiente OpenCL, passando pela construção e execução do *kernel*, até o retorno e atualização dos dados para a CPU.

Além de oferecer ganhos de desempenho, essa abordagem permite uma compreensão prática dos desafios e vantagens da programação heterogênea, promovendo um aprendizado sólido sobre o uso de GPUs em aplicações computacionalmente

intensivas. A estrutura modular do código facilita sua reutilização e expansão para projetos mais complexos, servindo como base para pesquisas futuras em simulações paralelas e computação de alto desempenho.

Portanto, este trabalho contribui não apenas como uma solução técnica funcional, mas também como um recurso didático relevante para estudantes e profissionais interessados em explorar os limites da computação paralela com OpenCL.

References

Uni.lu HPC School, 2021. Introduction to OpenCL Programming (C/C++). Disponivel em: https://ulhpc--tutorials-readthedocs-io.translate.goog/en/latest/gpu/opencl/?_x_tr_sl=en&_x_tr_tl=pt&_x_tr_hl=pt&_x_tr_pto=tc. Acesso em 09 de maio de 2025.

Menotti, Ricardo (2020), Programação Paralela: OpenCL (demo). Disponível em: https://youtu.be/f0QQdROR3Wo?si=3sB1g9og8L0_AKO5. Acesso em 09 de maio de 2025.