Programação paralela com Unity: um estudo de caso usando *threads* em CPU e em GPU

Vitor M. da Silva[1], **Rodrigo P. C. Nunes**[1], **Hélio Crestana Guardia**[1]

[1]Departamento de Computação – Universidade Federal de São Carlos (UFSCar)
Rod. Washington Luís, km 235 - Jardim Guanabara – São Carlos – SP – Brazil

{vitorsilva, rodrigo.coffani}@estudante.ufscar.br, helio.guardia@ufscar.br

Abstract. *This project is based on a graphical processing experiment, which involves moving spheres and calculating their positions in a scene using three different methods: sequentially, using multiple CPU threads, and using fine granularity parallelism for the GPU. As a result, this article aims to not just explore the performance gains with parallel processing in computing games, but also to link aspects of parallelism to visual elements that help understand parallelization and contribute to spark interest in the area of high-perfomance computing.*

Resumo. *Esse projeto é baseado em um experimento de processamento gráfico que envolve a movimentação e o cálculo das posições de esferas em uma cena usando três métodos diferentes: sequencial, usando múltiplas threads em CPUs, e com GPU para paralelismo com granularidade fina. Como resultado, este trabalho busca não apenas explorar os ganhos de desempenho com o processamento paralelo num cenário de jogos computacionais, mas também mapear aspectos do paralelismo a elementos visuais que colaborem para o entendimento da paralelização e contribuam para despertar o interesse para a área de computação de alto desempenho.*

1. Introdução

No desenvolvimento de jogos computacionais, o processamento de imagens requer grande poder de processamento de forma a prover uma boa experiência para o usuário. Este processamento, muitas vezes, trata-se da aplicação de um mesmo conjunto de operações sobre múltiplas partes de estruturas de dados relacionadas ao armazenamento e à representação de imagens. Como resultado, o uso de processamento paralelo, seja com estratégias que explorem a existência de múltiplos processadores (ou núcleos) ou mesmo GPUs para propósito geral, tem potencial para proporcionar ganhos de desempenho significativo nessas operações.

Assim, o objetivo deste trabalho é mostrar que o mesmo modelo de particionamento de tarefas com *threads* em CPU pode ser aplicado na programação paralela em cenários relacionados à manipulação de imagens em jogos, sendo uma oportunidade de aprendizado possibilitada por um nova perspectiva. A plataforma Unity [Unity 2024], amplamente utilizada em aplicações de jogos e em manipulações gráficas, também oferece suporte para a programação com múltiplas tarefas em CPU, assim como em GPU [Halladay 2014], em um modelo que assemelha-se com a definição de blocos de *threads* na programação com CUDA [Harris 2017]. Os resultados obtidos demonstram a viabilidade do uso de programação paralela na otimização de código no desenvolvimento de jogos.

2. Programação paralela na Unity: *Jobs* & *ComputeShaders*

Mesmo a Unity sendo muito popular para o desenvolvimento de jogos, o uso de conceitos de programação paralela no desenvolvimento de jogos com esta plataforma não recebe a devida importância, deixando a cargo do motor gráfico explorar adequadamente os mecanismos de alto desempenho disponíveis.

Contudo, assim como acontece com a programação com OpenMP, que permite a paralelização de código com o uso de múltiplas *threads*, ao programar-se com Unity, pode-se explorar os recursos da biblioteca de *Jobs*. Além disso, também é possível utilizar a GPU para programação paralela de propósito geral com Unity, utilizando a ferramenta de *Compute Shaders*, que opera num modelo semelhante ao proposto com a solução Nvidia CUDA [Harris 2017].

Para explorar esses modelos de programação paralela, este trabalho apresenta um estudo de caso baseado na movimentação das imagens de objetos 3D (esferas) na tela, avaliando os tempos envolvidos e a taxa de atualização das imagens resultantes, usando o processamento sequencial e códigos paralelos com múltiplas *threads* em CPU e em GPU.

3. Desenvolvimento

Por conta da Unity se tratar de um motor gráfico, a simples medição de tempo de cálculo da posição das esferas no estudo aqui realizado, não seria a mais adequada para entender os efeitos da capacidade de processamento e do uso do paralelismo. Assim, para medir o desempenho de cada método foi utilizada a métrica *FPS*[1], representando a taxa de atualização de quadros. Contudo, o quadro só é atualizado uma vez que todos os cálculos terminarem. Ou seja, quanto maior for a capacidade de processamento, maior o número de cálculos que poderão ser realizados antes que o desempenho seja afetado.

O experimento realizado para comparar os resultados dos diferentes modos de execução trata-se de esferas movendo-se por uma cena. Uma vez que a movimentação realizada pelas esferas é apenas vertical, o cálculo da posição feito por cada uma possui um custo computacional muito baixo, sendo necessário utilizar um grande número de esferas na tela antes que o desempenho seja afetado, o que dificultaria a análise visual dos resultados, principal aspecto a ser explorado pelo estudo. Assim, além dos cálculos de posicionamento e tratamento das imagens, inseriu-se nas operações um laço de repetição contendo cálculos mais complexos, com a quantidade de iterações desse *loop* sendo decidida em tempo de execução. Dessa forma, nos experimentos é possível aumentar o custo computacional associado à manipulação de cada esfera, sem alterar o número de esferas exibidas em tela, permitindo explorar a diferença entre os diferentes métodos sem prejudicar a experiência visual.

Como a performance depende do hardware, na configuração utilizada[2] para este estudo usou-se 4096 esferas, divididas em 16 blocos de 16x16, para a melhor exibição dos efeitos de cada método de processamento. A decisão da quantidade de esferas e sua divisão em blocos foi feita de acordo com a quantidade de recursos alocados para cada processo. Como no processamento realizado pela GPU são alocados blocos de *threads*, essa divisão facilita a visualização da tarefa que está sendo realizada.

[1] Frames per Second - Quadros por Segundo

[2] Notebook com processador Intel i5 13450HX e placa de vídeo RTX 3050 6GB

4. Resultados e Discussão

Para o método sequencial, que serviu de caso base, o cálculo das movimentações de todas as esferas é realizado por apenas uma *thread*. Como o *loop* de iterações é feito para cada esfera, o aumento de iterações aumenta o custo computacional para este método. O código resultante é apresentado na listagem a seguir.

```
1  public void Sequential(){
2   for(int i = 0; i < spheres.Length; i++) {
3     for (int j = 0; j < iterations; j++) // Perform extra calculations
4       spheres[i].calc += (int)(Mathf.Sqrt(5000)*(spheres[i].yPosition));
5     if(spheres[i].yPosition >= spheres[i].maxY) //Checks boundaries
6       spheres[i].speed = (((i)%64)+1)* -0.2f;
7     else if(spheres[i].yPosition <= spheres[i].minY)
8       spheres[i].speed = (((i)%64)+1)* 0.2f;
9     spheres[i].yPosition += spheres[i].speed * Time.deltaTime;
10  }
11 }
```

Ao aplicar o sistema de *Jobs* da Unity, o tratamento é muito similar aos mecanismos de OpenMP, utilizando os múltiplos processadores lógicos e suas respectivas *threads* para divisão dos cálculos. Neste estudo, tendo 10 processadores e 16 *threads*, pode-se atribuir cada fileira de esferas para uma *thread*, alterando a velocidade das esferas de acordo com o índice lógico de cada *thread*. Desta forma, foi possível observar que, a cada 16 linhas, o cálculo de posição é o mesmo, pois se tratava da mesma *thread*. O código a seguir ilustra o uso de *threads*. Na linha 4, vê-se a divisão do trabalho (manipulação das posições das esferas).

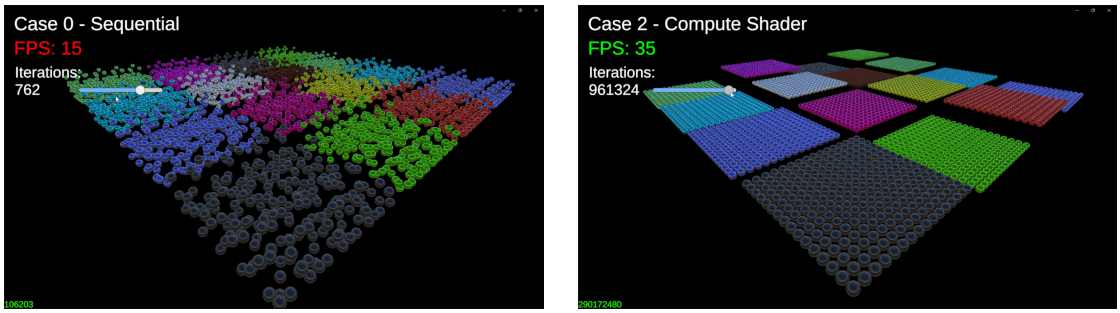
```
1  public void ConfigureJob(){
2    job = new SphereMoveJob(){...};
3    // Divide a manipulacao das esferas entre as threads
4    JobHandle jobHandle = job.Schedule(sphereList.Length, sphereList.Length / (
       _numberOfThreads-1)); // -1 para desconsiderar a main
5    jobHandle.Complete(); }     // Espera pela conclusao do job
6
7  // Loop paralelo, similar ao uso de #pragma omp paralell for
8  public struct SphereMoveJob : IJobParallelFor{
9    Sphere sphere = spheres[index];
10   // Mesmos calculos feitos no codigo sequencial
11   sphere.yPosition += spheres[index].speed * time;
12 }
```

Para explorar o uso da GPU nos processamentos, foi utilizado o recurso *Compute Shader* que, similar a CUDA[Harris 2017], utiliza a divisão de *threads* em uma grade de blocos. Para efeito de visualização dos resultados, utilizamos 16 blocos com 16x16x1 *threads*, de forma que todas as imagens de esferas na tela fossem tratadas em paralelo, uma por *thread*. Na listagem a seguir, vê-se o envio de dados para a GPU, a ativação do código *kernel* e o código do *kernel* utilizado.

```
1  private void SendDataToGPU() {
2      spheresBuffer = new ComputeBuffer(spheres.Length, totalSize);
3      spheresBuffer.SetData(spheres);
4      computeShader.SetBuffer(0, Spheres, spheresBuffer);
5      computeShader.SetInt(Iterations, iterations);
6  }
7  private void ComputeShader(){ // Inicia calculos na GPU
8      computeShader.Dispatch(0, gridDimX, gridDimY, gridDimZ);
9      spheresBuffer.GetData(spheres);  // Obtem os dados da GPU
10 }
11 #pragma kernel CSMain
12 [numthreads(16,16,1)]
13 void CSMain (uint3 id : SV_DispatchThreadID, uint3 gid : SV_GROUPID, uint3 gidthread :
       SV_GroupThreadID){
```

```
14      int block = gid.x * gridDimy * gridDimz + gid.y * gridDimz + gid.z;
15      // Mesmos calculos dos outros metodos, mas apenas para 1 esfera
16      // ...
17      sphere.yPosition += sphere.speed * time;
18  }
```

Para comparação dos resultados, considerou-se a capacidade de execução de pelo menos 30 FPS. Como era esperado, a execução do código sequencial ficou abaixo do limite estipulado de 30 quadros por segundo ao aumentar-se a carga de processamento nos cálculos a partir de 600 iterações. Com a execução visando um ambiente com múltiplas *threads* e paralelização na CPU, o desempenho teve uma evolução considerável mantendo o nível satisfatório até 7000 iterações, mantendo-se a taxa de atualização de até 30 FPS.



(a) Desempenho instável **(b) Desempenho estável**

Figura 1. Execução do código sequencial e paralelizado em GPU

Com o código paralelizado utilizando a GPU com o mecanismo de *Shaders*, o desempenho permaneceu estável atingindo até 63 FPS. Foi possível incrementar as iterações até próximo de 1 milhão, mantendo-se o desempenho estável e com uma taxa de 35 FPS. Na Figura 1, vê-se os resultados das execuções. Em (a), obtida com o processamento sequencial, à medida que aumenta-se o número de iterações, o desempenho fica instável, não sendo possível manter a taxa mínima de 30 FPS. Já em (b), obtida com o processamento paralelo usando *shaders* em GPU, vê-se que mesmo aumentando significativamente o processamento das iterações ainda maném-se uma elevada taxa de quadros (35 FPS).

5. Conclusões

A programação com múltiplas *threads* na Unity é um caminho para o bom desempenho de aplicações que envolvem manipulações de gráficos com restrição de tempo real. Seu modelo de exploração de paralelismo é eficiente e em muito se assemelha às estratégias de programação com *threads* em OpenMP e com a invocação de *Kernels* CUDA. Os resultados obtidos validam a eficiência desta plataforma no tratamento de operações paralelas. O produto final do código código produzido pode ser encontrado repositório no GitHub e o exemplo de execução em vídeo também está disponível online.

Referências

Halladay, K. (2014). Getting started with compute shaders in unity. Accesso em 7 de abril de 2024.

Harris, M. (2017). An even easier introduction to cuda. Accesso em 7 de abril de 2024.

Unity, T. (2024). Unity manual - compute shader. Accesso em 7 de abril 2024.