Computação Paralela

Detetor de cantos de *Harris* com CUDA e OpenMP

Alexandre Rodrigues, 92993, Gustavo Morais, 92978

3 de junho de 2022

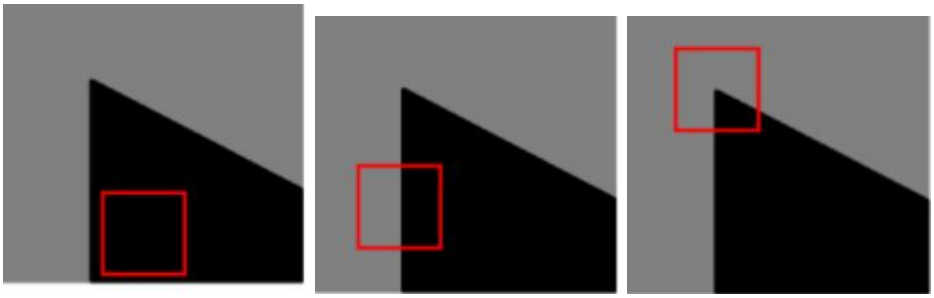
1 Introdução

1.1 Introdução ao detetor de cantos de *Harris*

O detetor de cantos de *Harris* é um operador de deteção de cantos que é normalmente utilizado em algoritmos de visão por computador para extrair cantos e inferir características de uma imagem. Foi introduzido pela primeira vez por *Chris Harris* e *Mike Stephens* em 1988 com o melhoramento do detetor de cantos de *Moravec* [1]. Comparado com o anterior, o detetor de canto de *Harris* tem em conta o diferencial da pontuação do canto com referência direta à direção, em vez de utilizar *patches* de deslocamento para cada ângulo de 45 graus, e provou-se ser mais preciso na distinção entre cantos e esquinas. Desde então, tem sido melhorado e adotado em muitos algoritmos para pré-processar imagens para aplicações subsequentes.

1.2 Cantos

Os cantos são características importantes de uma imagem, e são geralmente denominados como pontos de interesse que são invariantes à translação, rotação, e iluminação. Um canto pode ser interpretado como a junção de duas arestas, onde uma aresta é uma mudança súbita no brilho da imagem. Como podemos observar na figura 1(a), a região lisa não tem nenhuma alteração do gradiente em nenhuma das direções. De forma semelhante, na figura 1(b), não existe alteração do gradiente ao longo da borda. Já num canto, figura 1(c), existem alterações significativas do gradiente.



(a) (b) (c)

Figura 1: (a) Região lisa, sem mudanças em todas as direções; (b) Borda, sem mudanças ao longo da direção da borda; (c) Canto, mudanças significativas ao longo de todas as direções.

Posto isto, cantos são considerados características importantes graças à variação do gradiente em todas as direções ao contrário da região lisa e das bordas.

1.3 Deteção de cantos

Quando se tenta identificar cantos, a ideia é considerar uma janela à volta de cada pixel, p, que compõem uma imagem, tal como o quadrado vermelho presente na Figura 1. De forma a identificar os cantos, tomamos a diferença quadrática do pixel anterior e posterior, movemos a janela com o objetivo de identificar aquela onde a diferença quadrática é grande em todas as 8 direções. Definindo a função de mudança $E(u, v)$ como a soma de todas as diferenças quadráticas, onde u, v são as coordenadas x, y de cada pixel na janela e I a intensidade de cada pixel. As características na imagem são todos os pixeis que tem valores de $E(u, v)$ maiores que um certo limiar.

$$E(u,v) = \sum_{x,y} w(x,y)[I(x+u, y+v) - I(x,y)]^2 \tag{1}$$

Para detetar os cantos é necessário maximizar a função $E(u, v)$. Por outras palavras, é necessário maximizar o segundo termo. Aplicando alguns passos matemáticos obtemos a seguinte equação:

$$E(u,v) \approx \begin{bmatrix} u & v \end{bmatrix} \left(\sum \begin{bmatrix} I_x^2 & I_x I_y \\ I_x I_y & I_y^2 \end{bmatrix} \right) \begin{bmatrix} u \\ v \end{bmatrix} \tag{2}$$

Para simplificar a escrita:

$$M = \sum w(x,y) \begin{bmatrix} I_x^2 & I_x I_y \\ I_x I_y & I_y^2 \end{bmatrix} \tag{3}$$

Finalmente:

$$E(u,v) \approx \begin{bmatrix} u & v \end{bmatrix} M \begin{bmatrix} u \\ v \end{bmatrix} \tag{4}$$

Ao resolver para os vetores próprios de M, obtemos as direções tanto para os maiores como para os menores aumentos nas diferenças quadráticas. Os valores próprios correspondentes representam o valor destes aumentos. Uma métrica, R, é então calculada para cada janela:

$$R = \lambda_1 \lambda_2 - k(\lambda_1 + \lambda_2)^2 \tag{5}$$

onde λ_1 e λ_2 são os valores próprios de M. É através destes valores que se classifica uma região como lisa, borda ou canto

- Quando $|R|$ é pequeno, isto é, quando λ_1 e λ_2 são pequenos, a região é considerada lisa;
- Quando $R < 0$, o que acontece quando $\lambda_1 >> \lambda_2$ ou vice-versa, a região é considerada uma borda;
- Quando R é grande, o que acontece quando λ_1 e λ_2 são grandes e λ_1 λ_2, a região é um canto.

Figura 2: Resultado do algoritmo para a imagem *house.pgm*

2 Soluções desenvolvidas

2.1 OpenMP

A solução que desenvolvemos através do módulo OpenMP foi baseada num exercício desenvolvido na aula onde era, também, tratada uma imagem. Neste exercício cada *thread* executava alterações a uma parte da imagem paralelamente ás restantes. Aplicando a mesma lógica ao detetor de cantos de *Harris* desenvolvemos o código presente na figura 3.

```
#pragma omp parallel num_threads(10)
{
    int thIdx = omp_get_thread_num();
    int Nths = omp_get_num_threads();

    // define working regions
    int deltaI = floor(h/Nths);
    int minI = thIdx*deltaI;
    int maxI = minI + deltaI;

    for(int i=minI; i<maxI; i++) //height image
    {
        for(int j=0; j<w; j++) //width image
        {
            h_odata[i*w+j]=h_idata[i*w+j]/4; // to obtain a faded background image
        }
    }

    if (thIdx == Nths-1)
        maxI = h - ws - 1;
    if (thIdx == 0)
        minI = ws + 1;

    for(int i=minI; i<maxI; i++) //height image
    {
        for(int j=ws+1; j<w-ws-1; j++) //width image
        {
            sumIx2=0;sumIy2=0;sumIxIy=0;
            for(int k=-ws; k<=ws; k++) //height window
            {
                for(int l=-ws; l<=ws; l++) //width window
                {
                        Ix = ((int)h_idata[(i+k-1)*w + j+1] - (int)h_idata[(i+k+1)*w + j+1])/32;
                        Iy = ((int)h_idata[(i+k)*w + j+1-1] - (int)h_idata[(i+k)*w + j+1+1])/32;
                        sumIx2 += Ix*Ix;
                        sumIy2 += Iy*Iy;
                        sumIxIy += Ix*Iy;
                }
            }

            R = sumIx2*sumIy2-sumIxIy*sumIxIy-0.05*(sumIx2+sumIy2)*(sumIx2+sumIy2);

            if(R > threshold)
            {
                h_odata[i*w+j]=MAX_BRIGHTNESS;
            }
        }
    }
}
```

Figura 3: Zona paralelizada

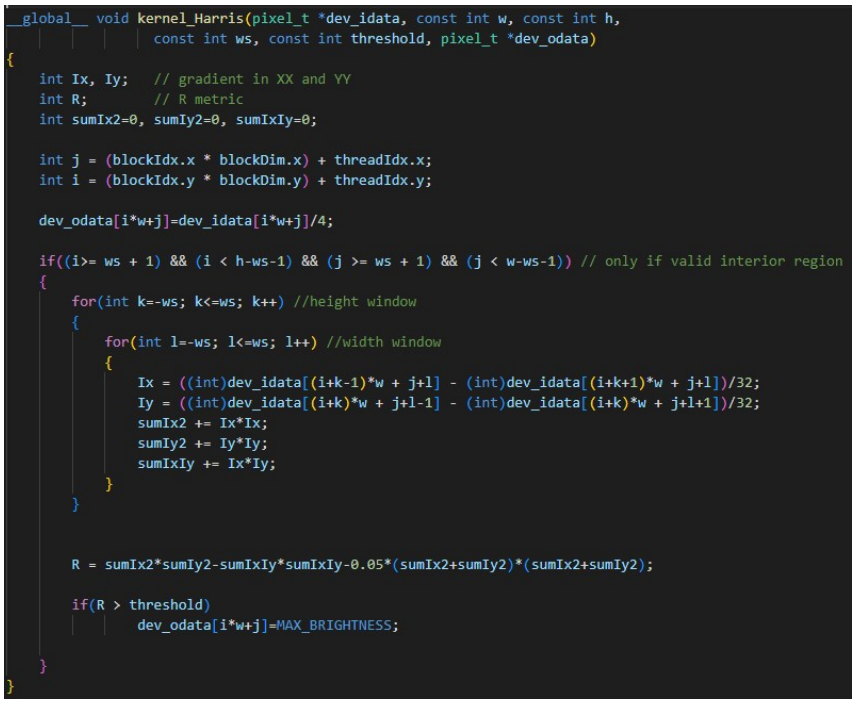
A diretiva *omp parallel* explicita ao compilador que o bloco de código seguinte vai ser paralelizado. Cada *thread* contida neste bloco, vai ter acesso ao seu número de identificação, $thIdx$, e o número total de *threads* lançadas, $Nths$. Estas variáveis em conjunto com a altura da imagem, h, vão permitir calcular o número de linhas que vão ser tratadas por cada *thread*, $deltaI$. De seguida, é calculado o índice inicial, $minI$, e o índice final, $maxI$, do ciclo que percorre as linhas da imagem. Devido ao uso de uma janela, o índice inicial da primeira *thread* e o índice final da ultima *thread* terão que ser alterados para garantir que não é acedida memoria exterior à imagem original.

2.2 CUDA

```
void harrisDetectorDevice(const pixel_t *h_idata, const int w, const int h,
                          const int ws, const int threshold,
                          pixel_t * h_odata)
{
    pixel_t *dev_idata, *dev_odata;
    pixel_t data_size;

    data_size = w*h*sizeof(pixel_t);
    printf("data_size %d\n", data_size);

    int numThreads_x = 32;
    int numThreads_y = 32;

    while((w % numThreads_x ) != 0)
        numThreads_x = numThreads_x - 1;

    while((h % numThreads_y ) != 0)
        numThreads_y = numThreads_y - 1;

    int numBlocks_x = ceil(w/numThreads_x);
    int numBlocks_y = ceil(h/numThreads_y);

    dim3 dimBlock(numThreads_x, numThreads_y); //  threadsPerBlock
    dim3 dimGrid(numBlocks_x, numBlocks_y); // numBlocks

    printf ("w = %d\n", w);
    printf ("h = %d\n", h);
    printf("dimBlock = %d x %d \n", dimBlock.x, dimBlock.y);
    printf("dimGrid = %d x %d \n", dimGrid.x, dimGrid.y);

    // memory allocation
    cudaMalloc((void **)&dev_idata, data_size);
    cudaMalloc((void **)&dev_odata, data_size);

    // copy image to device (CPU->GPU)
    cudaMemcpy(dev_idata, h_idata, data_size, cudaMemcpyHostToDevice);

    // Run corner detetion on GPU
    kernel_Harris<<<dimGrid, dimBlock>>>(dev_idata, w, h, ws, threshold, dev_odata);

    // Copy result from device to host (GPU->CPU)
    cudaMemcpy(h_odata, dev_odata, data_size, cudaMemcpyDeviceToHost);

    // free allocated memory
    cudaFree(dev_idata);
    cudaFree(dev_odata);
```

Figura 4: Implementação de CUDA

Tendo o objetivo de paralelizar o exercício com CUDA, começamos por definir o número limite de *threads* para as dimensões x e y, o número de blocos advém da divisão das dimensões da imagem a tratar pelo numero de *threads* de cada dimensão. Para garantir a divisão perfeita da imagem em *threads*, reduz-se o número de *threads* até à divisão não ter resto. Podemos então definir as variáveis *dimBlock* e *dimGrid* de forma semelhante aos exercícios das aulas, alocar a memória de imagens inicias e finais, copiar a imagem inicial para essa memória e correr a função `kernel_Harris` no GPU. O resultado será então copiado para a imagem final no CPU e a memória libertada.



```
__global__ void kernel_Harris(pixel_t *dev_idata, const int w, const int h,
                              const int ws, const int threshold, pixel_t *dev_odata)
{
    int Ix, Iy;   // gradient in XX and YY
    int R;        // R metric
    int sumIx2=0, sumIy2=0, sumIxIy=0;

    int j = (blockIdx.x * blockDim.x) + threadIdx.x;
    int i = (blockIdx.y * blockDim.y) + threadIdx.y;

    dev_odata[i*w+j]=dev_idata[i*w+j]/4;

    if((i>= ws + 1) && (i < h-ws-1) && (j >= ws + 1) && (j < w-ws-1)) // only if valid interior region
    {
        for(int k=-ws; k<=ws; k++) //height window
        {
            for(int l=-ws; l<=ws; l++) //width window
            {
                Ix = ((int)dev_idata[(i+k-1)*w + j+l] - (int)dev_idata[(i+k+1)*w + j+l])/32;
                Iy = ((int)dev_idata[(i+k)*w + j+l-1] - (int)dev_idata[(i+k)*w + j+l+1])/32;
                sumIx2 += Ix*Ix;
                sumIy2 += Iy*Iy;
                sumIxIy += Ix*Iy;
            }
        }


        R = sumIx2*sumIy2-sumIxIy*sumIxIy-0.05*(sumIx2+sumIy2)*(sumIx2+sumIy2);

        if(R > threshold)
                dev_odata[i*w+j]=MAX_BRIGHTNESS;

    }
}
```

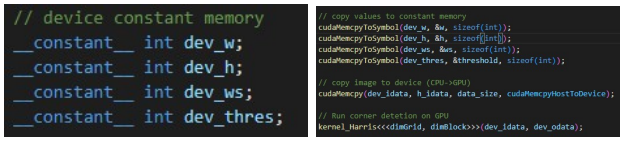
Figura 5: Kernel para a implementação com *Global Memory*

A função *kernel_Harris* correrá para cada *thread* de cada bloco, assim podemos calcular os índices para percorrer a imagem da seguinte forma:

```
j = (blockIdx.x * blockDim.x) + threadIdx.x;
i = (blockIdx.y * blockDim.y) + threadIdx.y;
```

A redução de intensidade da imagem é feita para todos os pixeis, no entanto os ciclos *for* de deteção de cantos são apenas validos para a zona interior da imagem (exclui as $ws + 1$ colunas/linhas da borda), tal condição é garantida com o *if* da linha 84.

Exploramos diferentes tipos de memória que poderiam ser úteis para o problema, a versão inicial usa *global memory* por ser mais simples de implementar. *Shared memory* seria um possibilidade mas não conseguimos encontrar uma implementação que beneficiasse este problema já que o resultado é independente entre *threads*. A implementação inicial de *texture memory* que tentamos produzir foi bem sucedida apenas para imagens quadradas (*chess* das imagens dadas). A melhor solução para o problema foi o uso de *constant memory* para guardar os valores de largura, altura, tamanho da janela e limiar (*w, h, ws, threshold*), resultando numa pequena aceleração adicional.



```
// device constant memory
__constant__ int dev_w;
__constant__ int dev_h;
__constant__ int dev_ws;
__constant__ int dev_thres;
```

(a)

```
// copy values to constant memory
cudaMemcpyToSymbol(dev_w, &w, sizeof(int));
cudaMemcpyToSymbol(dev_h, &h, sizeof(int));
cudaMemcpyToSymbol(dev_ws, &ws, sizeof(int));
cudaMemcpyToSymbol(dev_thres, &threshold, sizeof(int));

// copy image to device (CPU->GPU)
cudaMemcpy(dev_idata, h_idata, data_size, cudaMemcpyHostToDevice);

// Run corner detetion on GPU
kernel_Harris<<<dimGrid, dimBlock>>>(dev_idata, dev_odata);
```

(b)

Figura 6: Alterações para uso de *Constant Memory*

Na figura 6(a) vemos as declarações de variáveis em memória constante, feito nas linhas 28 a 31, antes da função `harrisDetectorHost`. A figura 6(b) refere-se às alterações na função `harrisDetectorDevice`, antes da execução do *kernel* e a remoção das variáveis passadas para o mesmo.

3 Como correr os programas

Para conseguirmos testar completamente os programas `harrisDetectorOpenMP.c` e `harrisDetectorCuda.cu` desenvolvemos, em *bash*, o programa `run.sh`. No programa `run.sh` utilizamos a função `getopts` com as seguintes opções:

- -c *img* : faz *make*, corre o programa `harrisDetectorCuda.cu` com *img* como imagem de entrada e, por fim, verifica as diferenças entre a imagem do *host* e a que sai do programa `harrisDetectorCuda.cu` (Global Memory)
- -m *img*: faz *make*, corre o programa `harrisDetectorCudaConst.cu` com *img* como imagem de entrada e, por fim, verifica as diferenças entre a imagem do *host* e a que sai do programa `harrisDetectorCudaConst.cu` (Constant Memory)
- -t *img*: faz *make*, corre o programa `harrisDetectorCudaTexture.cu` com *img* como imagem de entrada e, por fim, verifica as diferenças entre a imagem do *host* e a que sai do programa `harrisDetectorCudaTexture.cu` (Texture Memory)
- -o *img* : faz *make*, corre o programa `harrisDetectorOpenMP.c` com *img* como imagem de entrada e, por fim, verifica as diferenças entre a imagem do *host* e a que sai do programa `harrisDetectorOpenMP.c`
- -n N : tem de ser utilizado em conjunto com uma das opções anteriores. Permite a repetição do programa N vezes, mostra e guarda o tempo que demora cada execução e no fim faz uma média dos tempos

O programa `run.sh` apenas funciona se os ficheiros não se encontrarem em diretorias diferentes.

4 Sumário dos resultados

Algo que reparamos assim que concluímos a implementação de OpenMP é que nem sempre era útil utilizar as 24 *threads* disponíveis, nomeadamente devido ao *overhead* dos cálculos dos limites das regiões da imagem a ser tratadas por cada *thread*. Posto isto resolvemos fazer o estudo do número ótimo de *threads* a utilizar. Optamos por fazer apenas para as imagens *chess* e *chessBig* e podemos visualizar os resultados na figura 7. As linhas traçadas nos gráficos correspondem aos tempos médios de 100 execuções do programa `harrisDetectorOpenMP.c`. Daqui podemos concluir que para a imagem *chess* e para a imagem *chessBig*, o número ótimo de *threads* seria 8 e 11, respetivamente.

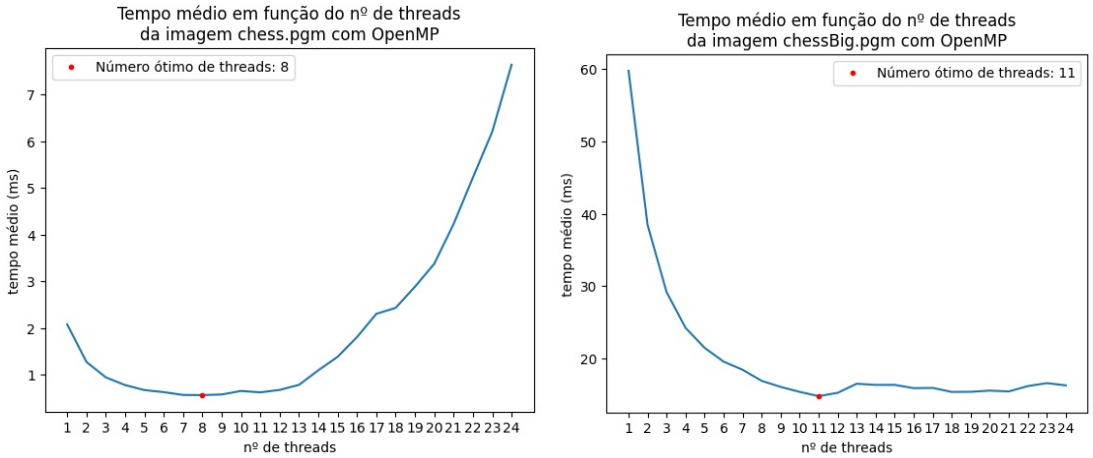


Figura 7: OpenMP: Tempo médio de execução em função do número de *threads* para as imagens *chess* e *chessBig*.

Tendo por base o número ótimo de *threads* apresentados para as imagens *chess* e *chessBig*, definimos como 10 o número de *threads* que iríamos utilizar para as restantes imagens. Os resultados correspondentes aos tempos de 100 execuções por imagem estão presentes no anexo 5, tanto para OpenMP como para CUDA e as suas diferentes memórias. Adicionalmente, um sumário dos resultados encontra-se na tabela 1 e os respetivos *speedups* na tabela 2.

Dispositivo / Imagem	Host	OpenMP	CUDA Global Memory	CUDA Constant Memory
chess	1.865	0.625	0.384	0.368
chessBig	97.448	15.042	4.74	4.622
chessL	2.857	0.78	0.471	0.448
chessRotate1	1.8	0.653	0.384	0.367
house	3.607	0.945	0.499	0.477

Tabela 1: Tempos médios de 100 execuções em milissegundos para as diferentes imagens e diferentes dispositivos.

Dispositivo / Imagem	OpenMP	CUDA Global Memory	CUDA Constant Memory
chess	2.984x	4.856x	5.067x
chessBig	6.478x	20.55x	21.08x
chessL	3.662x	6.065x	6.377x
chessRotate1	2.756x	4.687x	4.904x
house	3.816x	7.228x	7.561x

Tabela 2: Speedups médios de 100 execuções para as diferentes imagens e diferentes dispositivos.

Os resultados obtidos são de acordo com o esperado: OpenMP produz um *speedup* considerável mas CUDA tem uma performance superior. O uso de *constant memory* produz uma melhoria adicional. Pode-se também verificar que os efeitos são maiores para a imagem *chessBig* dado que o processamento em CPU é já de duração considerável.

Referências

[1] C. Harris and M. Stephens, "A Combined Corner and Edge Detector," *4th Alvey Vision Conference, Manchester, UK*, p. 147–151, 1988.

5 Anexos

5.1 Anexo OpenMP

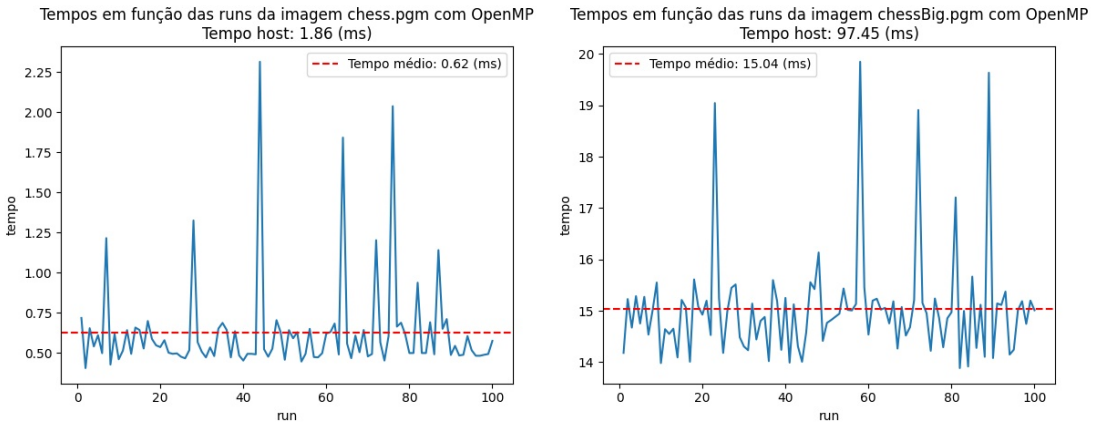


Figura 8

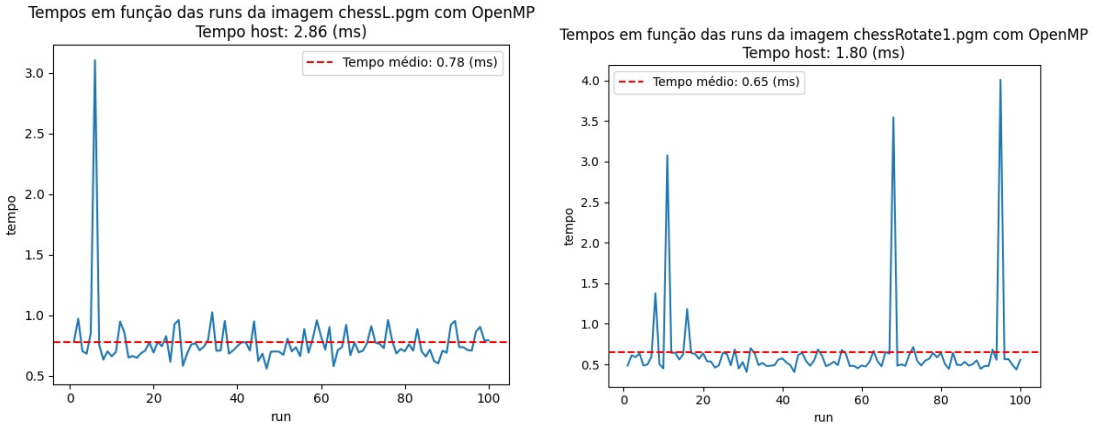


Figura 9

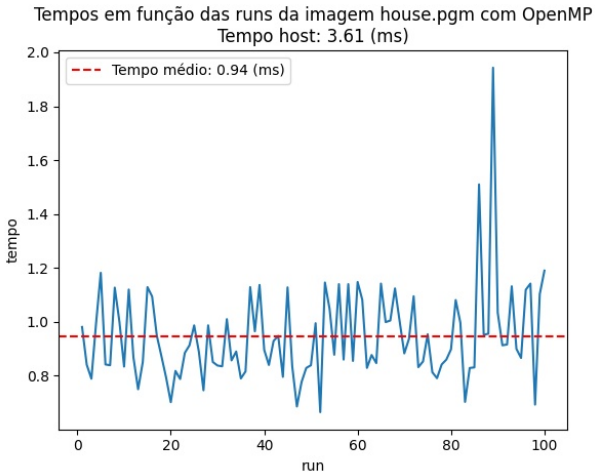


Figura 10

5.2 Anexo CUDA

5.2.1 Global Memory

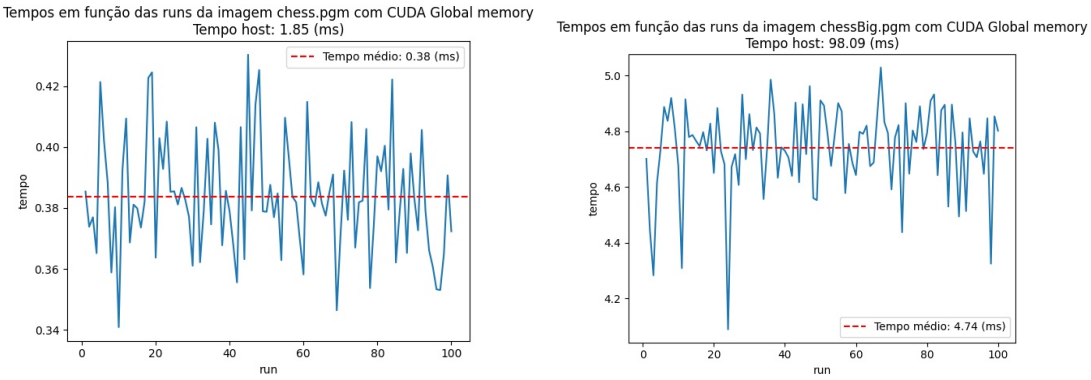


Figura 11

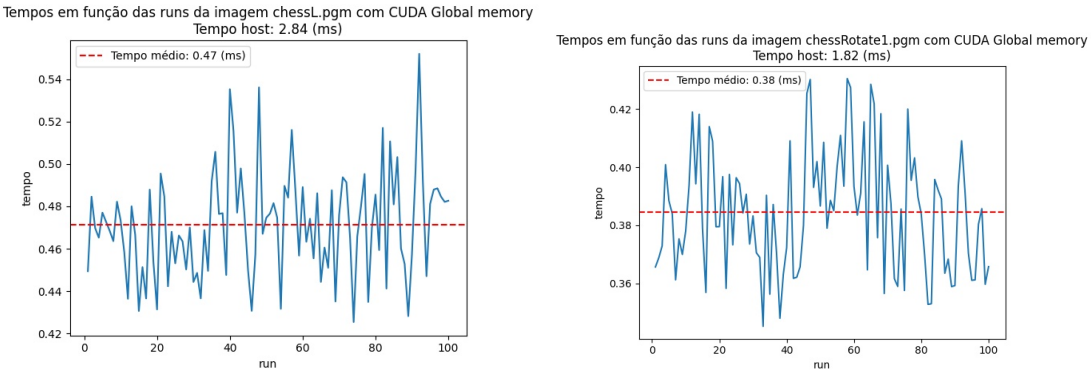


Figura 12

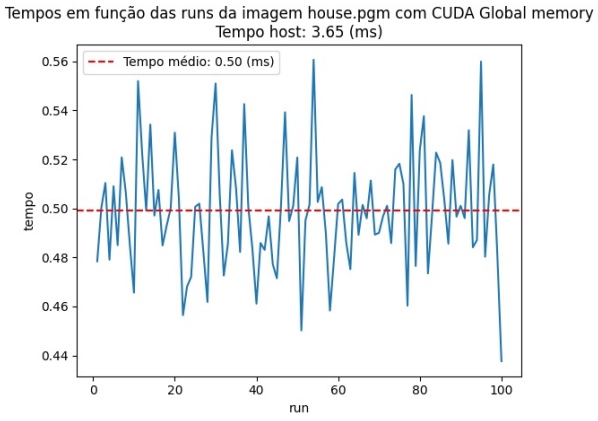
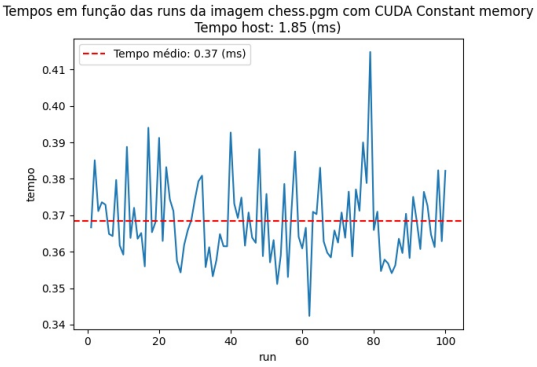


Figura 13

5.2.2 Constant Memory



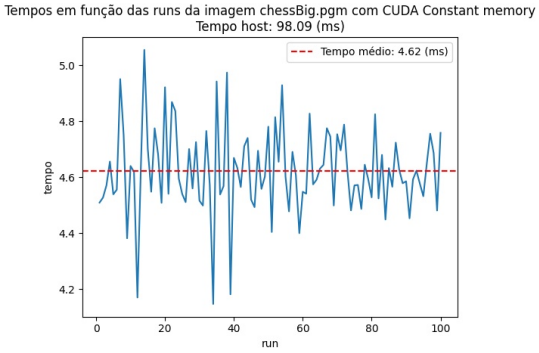
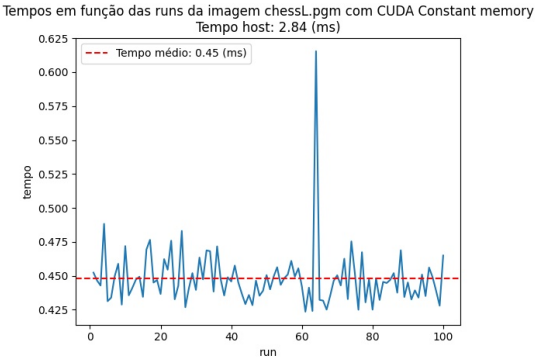


Figura 14



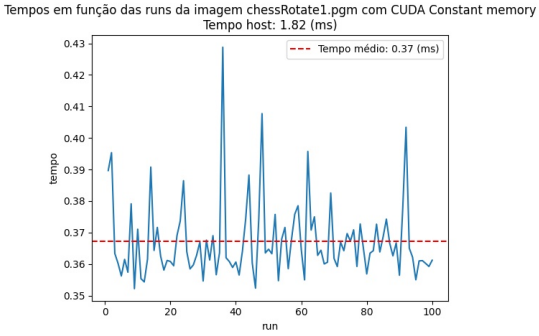


Figura 15

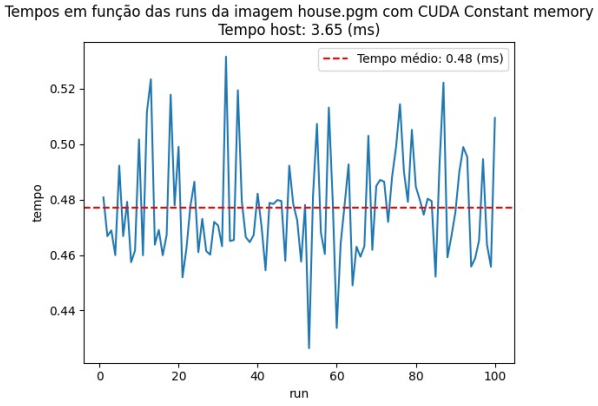


Figura 16