Computação Paralela

Projeto 1

Harris Corner Detection with CUDA and OpenMP

Álvaro Freixo, 93116
Mestrado em Engenharia Computacional

Rita Ferrolho, 88822
Mestrado em Engenharia Computacional

2021-22

1 Introdução

Em 1961, com o lançamento do primeiro sistema informático multi tarefa, *Leo III*, proporcionou o nascimento da era da Computação Paralela na área da tecnologia. A multitarefa preempetiva foi implementada, em 1969, nas versões embrionárias do *UNIX*, padronizando este e outros sistemas, tais como *Linux*, *Solaris* e *BSD*. Posterior à emulação de multitarefa, nas versões primitivas do *Windows* baseadas em *DOS* (sistema monotarefa), surgiu o escalonador de processos (*scheduler*), uma das principais componentes dos sistemas operativos atuais.

Atualmente, na era da informação, a paralelização de processos é fundamental. Permite um melhor aproveitamento de recursos, melhor eficiência e rapidez de processamento sendo possível distribuir tarefas locais e/ou por vários sistemas.

No presente trabalho é proposto a adaptação de um programa sequencial em dois métodos distintos de paralelização, *OpenMP* e *CUDA*. O programa mencionado processa imagens em escala de cinzentos de modo a detetar a posição dos cantos, Fig. 1. Os cantos de uma imagem são importantes, sendo denominados pontos de interesse que são invariantes à translação, rotação e iluminação, Fig. 2. O programa é baseado no *Harris Detector*, o qual processa os gradientes e a soma do produto dos mesmos numa janela em torno do pixel considerado. O algoritmo calcula a métrica *R* baseada nos valores próprios da matriz. Caso *R* seja elevado (valores próprios também elevados), então será um canto, Fig. 3.

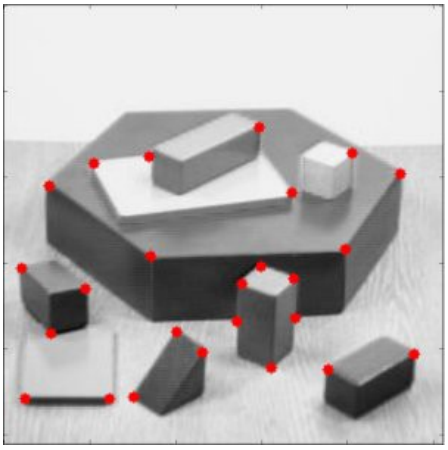


Fig. 1: Exemplificação do resultado a obter.

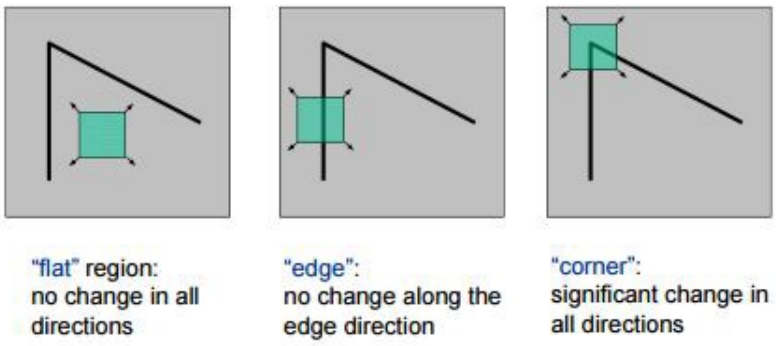


Fig. 2: Caracterização dos cantos sendo a junção de duas arestas.

Os métodos mencionados distiguem-se, fundamentalmente, no hardware de paralelização, CPU (*OpenMP*) e GPU NVIDIA (*CUDA*). É visível a elevada densidade de núcleos de processamento nas GPUs (10752 *CUDA Cores*, Geforce RTX 3090 Ti) em comparação aos CPUs (128 threads, AMD EPYC 7773X). Este facto é explicado pelos objetivos distintos de processamento, sendo que as GPUs são mais indicadas para processamento repetitivo com o mesmo código a ser paralelizado nos diversos núcleos, por exemplo, renderização de Jogos, treinar Redes Neuronais, etc.

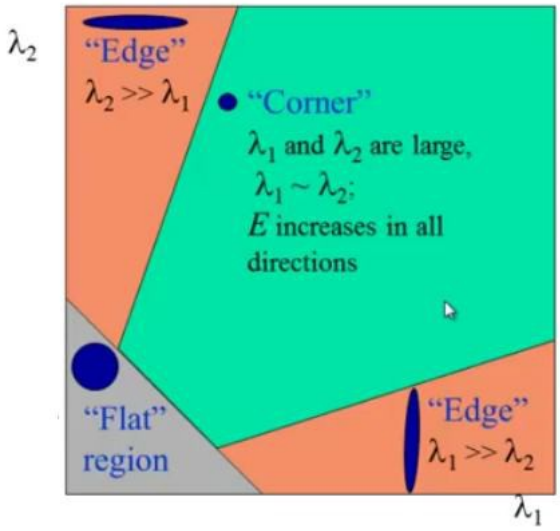


Fig. 3: Relação entre valores próprios (*R*) e caracterização de cantos.

2 OpenMP

Open Multi-Processing, publicado em outubro de 1997 pelo Conselho de Análise e Arquitetura (Architecture Review Board, ARB) para *FORTRAN 1.0* e, em outubro de 1998, para C/C++, é uma *API* (application programming interface) caracterizada pelo modelo de programação de memória partilhada destinada a multiprocessadores e arquiteturas *multicore*.

Os objetivos desta especificação é a portabilidade do model para arquiteturas de memória partilhada; simplicidade das diretivas de programação; possibilidade de paralelismo incremental para programas sequenciais; eficiência de implementações para diversos problemas.

2.1 Metodologia

O algoritmo de *Harris Detector* disponibilizado foi adaptado recorrendo à implementação 1. De notar que as linhas 1 e 10 foram introduzidas para o efeito. Foram elaboradas diversas versões de modo a comparar a eficiência das alterações introduzidas para a imagem "chessBig.pgm" devido a uma maior discrepância entre os tempos de execução sequencial e paralelizado para facilitar a análise, Tabela I. A implementação foi executado recorrendo a 4 threads de um PC portátil com Ubuntu em *bare metal* com um CPU *Intel i7 7500U*.

```
1   #pragma omp parallel for shared(h_odata) private(i, j) firstprivate(h_idata, w, h) schedule
     (guided,8) collapse(2)
2    for(i=0; i<h; i++) //height image
3    {
4        for(j=0; j<w; j++) //width image
5        {
6            h_odata[i*w+j]=h_idata[i*w+j]/4; // to obtain a faded background image
7        }
8    }
9
10   #pragma omp parallel for shared(h_odata) private(j, sumIx2, sumIy2, sumIxIy, Ix, Iy)
     firstprivate(h_idata, w, h, ws, threshold) schedule(guided,8) collapse(2)
11   for(i=ws+1; i<h-ws-1; i++) //height image
12   {
13       for(j=ws+1; j<w-ws-1; j++) //width image
14       {
15          sumIx2=0;sumIy2=0;sumIxIy=0;
16          for(k=-ws; k<=ws; k++) //height window
17              {
18                  for(l=-ws; l<=ws; l++) //width window
19                  {
20                    Ix = ((int)h_idata[(i+k-1)*w + j+l] - (int)h_idata[(i+k+1)*w + j+l])/32;
21                    Iy = ((int)h_idata[(i+k)*w + j+l-1] - (int)h_idata[(i+k)*w + j+l+1])/32;
22                    sumIx2 += Ix*Ix;
23                    sumIy2 += Iy*Iy;
24                    sumIxIy += Ix*Iy;
25                  }
26              }
```

```
27
28              R = sumIx2*sumIy2-sumIxIy*sumIxIy-0.05*(sumIx2+sumIy2)*(sumIx2+sumIy2);
29              if(R > threshold) {
30                   h_odata[i*w+j]=MAX_BRIGHTNESS;
31              }
32         }
33     }
```

Listing 1: Harris Detector, OpenMP, C/C++, implementação na função *harrisDetectorOpenMP*

As versões mencionadas na tabela I evidenciam diversos tipos de interações entre threads, *OpenMP schedules*.

- schedule(static,8): O *OpenMP* divide as iterações em 8 pedaços distribuídos em ordem circular. Um possivel exemplo é paralelizar um ciclo de 64 iterações, em que cada linha da Fig. 2 representa cada thread e, cada coluna uma iteração. Em teoria, é mais indicado quando cada iteração tem o meu custo/dificuldade de processamento.

```
1 ********                          ********
2         ********                          ********
3                 ********                          ********
4                         ********                          ********
5
```

Listing 2: Representação gráfica de *schedule(static,8)*

- schedule(dynamic, 8): Divisão das iterações em 8 pedaços não existindo uma ordem específica de organização dos threads temporalmente, Fig, 3. É aconselhado para quando as iterações têm custos de processamento diferentes. Existe alguma lentidão comparativamente ao *static* devido a distribuição dos threads durante a execução (*runtime*).

```
1                 ********                                    ********
2                         ********          ********
3 ********                          ********          ********
4         ********
5
```

Listing 3: Representação gráfica de *schedule(dynamic, 8)*

- schedule(guided, 8): Parecido ao *dynamic*, cada thread executa um pacote de iterações pedindo outro pacote até não haver mais disponíveis. A diferença reside no facto do tamanho dos pacotes serem proporcionais ao número de iterações não atribuídas dividido pelo número dos threads, existindo um decréscimo do tamanho dos pacotes de iterações. o Tamanho mínimo dos pacotes é definido na função, neste caso seria 8, mas pode ser inferior a esse valor. É aconselhado quando existe um mau balanceamento de dificuldade computacional ao longo da execução do programa, por exemplo, este poderá aumentar ao longo do tempo de execução. Fig. 4.

```
1                 ************                    ********          ***
2 ****************
3                                     ********
4                             *********                 ********
5
```

Listing 4: Representação gráfica de *schedule(guided, 8)*

- collapse(2): Possibilita a paralelização de *nested loops*. Neste caso, em vez de paralelizar somente o ciclo *for* exterior, divisão das linhas da matriz representativa da imagem por cada thread de acordo com os *schedules*, é possível atribuir individualmente os pixels a cada thread. Uma restrição necessária à implemnetação é a independência do cálculo dos dados de cada *loop*. É aconselhável quando o *loop* exterior não tiver menos iterações que a quantidade de threads disponíveis.

As variáveis utilizadas na implementação dividem-se em três categorias, *shared*, *private* e *firstprivate*. A primeira engloba os arrays input e output da imagem a processar (*h_odata, h_idata*), a qual deverá ser partilhada por todos os *threads* a fim de uma melhor eficiência de memória, caso o acesso a esta seja bem executado de modo a evitar concorrência. As variáveis do *private* são necessárias para alocar a cada *thread* variáveis incrementáveis durante 'loops' (por exemplo, *i, j, sumIx2, sumIy2, sumIxIy, Ix, Iy*). As *first private* são equivalente às *private*, mas utilizando o valor inicializado fora da zona de paralela (iniciada pela invocação *#pragma omp parallel for*).

2.2 Análise de Resultados

Os resultados das variações da implementação foram comparadas com os resultados sequenciais para todas a imagens disponíveis utilizando o script fornecido (*testDiffs*), O script reportou zero diferenças.

Para facilitar a análise foram somente comparados os tempos de execução para a imagem *chessBig.pgm*.

Na tabela I é possível verificar que o tempo final sequencial tem alguma variação dependendo do estado em que se encontra o CPU e devido às execuções consecutivas apesar de ter sido usado um intervalo de tempo de 10 segundos.

É possível verificar o *static 8* tem uma boa eficiência, pois, neste projeto, não existe uma grande discrepância de dificuldade entre *threads* ao longo da execução. De certa forma, o *guided 30* é melhor que os restantes *guided* pois tenderá a ser mais próximo do *static* existindo threads com mais interações alocadas inicialmente diminuindo alguma possível latência de cálculo de alocação de iterações. O *guided 30* em relação ao *static*, também, se comporta melhor devido à melhoria de balanceamento reduzindo algum possível atraso (mínimo) de algum *thread* no cálculo.

As variações com *collapse* têm, em média, uma menor eficiência devido ao *dynamic*. Neste caso, excluindo a variação mencionada, até teve um desempenho equiparável aos restantes, mas poderia ter sido superior caso tivesse sido executado uma máquina com maior quantidade de *threads*.

O *dynamic* foi o pior com e sem *collapse* devido à elevada latência na alocação dos pacotes de iterações a cada *thread*.

Implementação	Tempo final Sequencial	Tempo final Paralelização	Eficiência %
schedule(guided,8)	72.391	34.336	52.57%
schedule(guided,2)	75.074	34.119	54.55%
schedule(guided,30)	80.364	33.931	57.78%
schedule(static,8)	78.884	34.566	56.18%
schedule(dynamic,8)	73.466	35.415	51.79%
schedule(guided,8) collapse(2)	75.854	32.77	56.80%
schedule(guided,2) collapse(2)	77.03	34.603	55.08%
schedule(guided,30) collapse(2)	76.179	33.069	56.59%
schedule(static,8) collapse(2)	79.066	40.598	48.65%
schedule(dynamic,8) collapse(2)	77.259	44.975	41.79%

Table I: OpenMP - Tabela tempos finais. Imagem *chessBig.pgm*, 4 *Threads*, execuções consecutivas, 10s de espera entre execuções. *testDiffs* reportou zero diferenças. Eficiência é a percentagem de poupança de tempo.

3 CUDA

Compute Unified Device Architecture, publicado em 2006 pela fabricante de placas gráficas *NVIDIA*, é uma plataforma paralela e uma API (application programming interface), que possibilita o uso de placa gráficas *NVIDIA* para processamento genérico (general-purpose computing on GPUs, GPGPU).

Neste paradigma de processamento, existe a noção de *kernel*. É caracterizado por ser um mini-programa ou uma subrotina que executa em paralelo num *device* (uma placa gráfica NVIDIA).

Na constituição de um *kernel* existe uma *grid*, a qual engloba vários *thread blocks* de igual dimensão executados no mesmo *kernel*. As *grids* são úteis para processar um elevado número de *threads* em paralelo devido ao facto da limitação de 512 *threads* em cada *thread block* consequência da limitação de memória partilhada (*shared memory*), também, em cada *thread block*. Os *threads* de cada *thread block* podem comunicar eficazmente entre si através da *shared memory* podendo, também, garantir a sua sincronização (*sync*). Em oposição, *thread blocks* não podem comunicar entre si pela *shared memory* e, consequentemente, mão existe possibilidade de sincronização entre si. Conceptualmente as *grids* e os *thread blocks* estão organizados em arrays de uma até 3 dimensões por conveniência. Essa organização possibilita a identidicação dos *thread blocks* e dos *threads*, Fig. 4.

Numa placa gráfica (GPU) existem diversos níveis de memória, cada uma com diferentes características de leitura e escrita. Cada *thread* tem acesso a uma memória local privada, alojada na memória global GDDR disponível na GPU. Cada *thread* num *thread block*, também, tem acesso à *shared memory* partilhada por todos os *threads* do bloco. Todos os *threads* da *grid* têm acesso à *global memory*, que está localizada fora do chip principal da memória GDDR (o qual é o mais volumoso, mas o aceso é mais lento). Também existem memórias do tipo *read-only*, *constant* e *texture* na mesma localização da *global memory*.

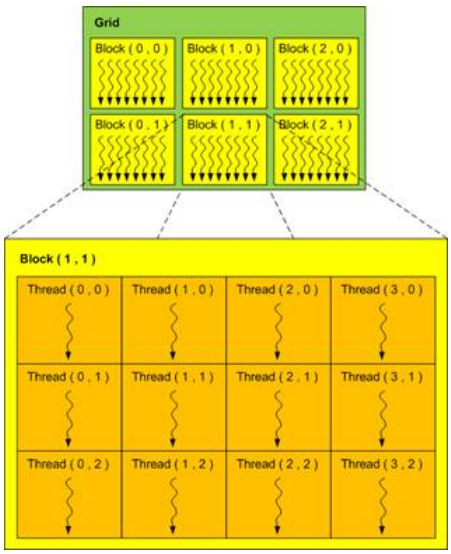


Fig. 4: Representação gráfica da hierarquia da constituição e organização de *thread blocks*, *threads* e *grid*. Neste caso, o *Kernel* contem uma *grid* 3x2 de *thread blocks*. Cada *thread blocks* tem um bloco 4x3 de *threads*, ao todo existem 72 *threads* a serem executados no dito *kernel*

Cada tipo de memória é otimizada para diferentes utilizações. No caso da *global memory*, não tem *cache* sendo mais lenta. A *constant memory* é *read-only*, reside na mesma localização da *global*, mas tem *cache* associado. A *texture memory*, também, é *read-only*, residindo na mesma localização da *global* tendo *cache*. Esta difere da *constant* pela política da *cache* explorando localizações a duas e três dimensões, Fig. 5.

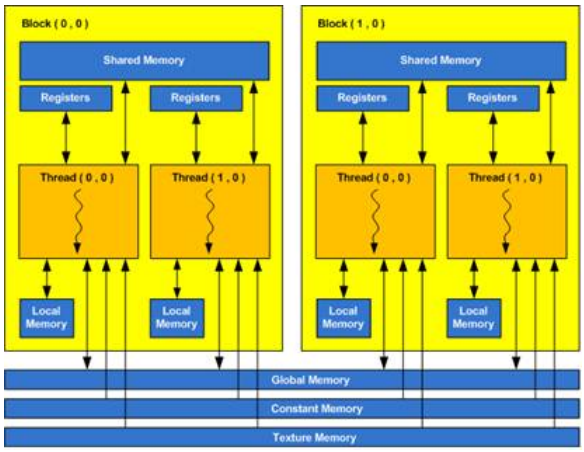


Fig. 5: Representação gráfica da hierarquia de segmentos de memória em CUDA

Existe uma panóplia de models de placas gráficas, neste caso, da fabricante NVIDIA, compatíveis com CUDA. Deste modo, cada models tem especificações distintas e poderão ter suporte a diferentes *features* e recursos computacionais. Quando um *kernel* invocado, este é executado num *multiprocessor*, o qual tem recursos para suportar um certo número de *threads*. Um ou mais *thread blocks* são alocados para um *multiprocessor* durante a execução do *kernel*. O *CUDA runtime* é responsável pela *dynamic scheduling* dos *thread blocks* a um grupo de *multiprocessors*. Um *thread blocks* será alocado a um *multiprocessor* se existirem recursos para tal. Cada bloco é dividido em grupos SIMD (*Single-Instruction Multiple-Data*), chamados *warps*. A unidade SIMD gerencia 32 *threads* em simultâneo, para criar um *warp*. Cada *warp* é síncrono e, deve ser assegurado que os *threads* têm um bom balanceamento de dificuldade de execução, pois a rapidez do *warp* é limitada pelo *thread* mais lento.

Todas as placas gráficas que suportam CUDA têm um número que determina a disponibilidade de *features* a nível de hardware, denominado de *Compute Compatibility*. Por exemplo, a NVIDIA Tesla C870 tem a verão 1.0 de *Compute Compatibility*, enquanto a NVIDIA Geforce RTX 3090 suporta até à versão 8.6.

Neste projeto, foi utilizado um servidor com uma GPU Geforce 1660 Ti, com *Compute Compatibility* limitado à versão 3 (esta GPU é compatível até à versão 7.5) e um AMD Ryzen 9 3900X.

3.1 Global Memory

3.1.1 Metodologia

A implementação elaborada para CUDA em *global memory*, Fig. 5, inclui a inicialização de variáveis relativas à organização da *grid* (linha 5), dos *thread blocks* (linha 4), promove a existência de um *warp* em cada *block* e, variável (linha 3) para futura alocação de memória/endereçamento na GPU dos *arrays* em memória global (linhas 12 e 13). O envio dos arrays de input e output da imagem a processar para a GPU (linhas 17 e 18). As dimensões da *grid* e dos *blocks* foram definidos nas linhas 22 e 23. A sua organização no eixo x (horizontal) assemelha o *array* input e output da imagem (*h_idata*, *h_odata*). A invocação do *kernel* na linha 26 incluindo os parâmetros de entrada da função do kernel da Fig. 6. Neste caso, a invocação do *kernel* acontece duas vezes, em conjunto com a variável *kernel_type*, a primeira para fazer o *fade* da imagem (linha 15, Fig.6; linha 26, Fig.5), a segunda para calcular os cantos (Linha 22 a 51, Fig.6; linha 34; Fig.5) utilizando a imagem *faded* já em memória global.

Na função *reduce1* Fig.6, nas linhas 20 e 21, é possível identificar a linha e a coluna do pixel que corresponde ao *thread* para ser utilizado (através da linha *i* e coluna *j*) no processamento dos cantos da linha 30 à 49. A condição *if* da linha 34 evita o processamento das bordas da imagem na deteção dos cantos.

```
1     int size = h * w;
2     int kernel_type = 0;
3     int memsize = size * sizeof(pixel_t);
4     int threadsPerBlock = 32;
5     int blocksPerGrid = size / threadsPerBlock;
6
7     // allocate host memory
8     pixel_t *devPtrh_idata;
9     pixel_t *devPtrh_odata;
10
11    // Allocate device memory
12    cudaMalloc((void **)&devPtrh_idata, memsize);
13    cudaMalloc((void **)&devPtrh_odata, memsize);
14    // cudaMalloc((void **)&devPtrsize, memsize_size_arr);
15
16    // Copy data (data to process) from host to device (from CPU to GPU)
17    cudaMemcpy(devPtrh_idata, h_idata, memsize, cudaMemcpyHostToDevice);
18    cudaMemcpy(devPtrh_odata, h_odata, memsize, cudaMemcpyHostToDevice);
19
20    // Call kernel to Fade image
21    // __global__ functions are called:  Func <<< dim grid, dim block >>> (parameter);
22    dim3 dimGrid(blocksPerGrid, 1, 1);
23    dim3 dimBlock(threadsPerBlock, 1, 1);
24
25    // Execute the Kernel
26    reduce1<<<dimGrid, dimBlock>>>(devPtrh_idata, devPtrh_odata, kernel_type, ws, w, threshold
      , h, size);
27
28    // Copy data from device (results) back to host
29    // cudaMemcpy(h_odata, devPtrh_odata, memsize, cudaMemcpyDeviceToHost);
30
31    kernel_type = 1;
32    // Call kernel to calculate corners
33    // __global__ functions are called:  Func <<< dim grid, dim block >>> (parameter);
34    reduce1<<<dimGrid, dimBlock>>>(devPtrh_idata, devPtrh_odata, kernel_type, ws, w, threshold
      , h, size);
35
36    // Copy data from device (results) back to host
37    cudaMemcpy(h_odata, devPtrh_odata, memsize, cudaMemcpyDeviceToHost);
38
39    // Free device memory
40    cudaFree(devPtrh_idata);
41    cudaFree(devPtrh_odata);
```

Listing 5: Harris Detector, Cuda Global memory, C/C++, implementação na função *harrisDetectorDevice*

```
1  __global__ void reduce1(pixel_t *h_idata, pixel_t *h_odata, int kernel_type, int ws, int w,
       int threshold, int h, int size)
2  {
3      unsigned int id = blockIdx.x * blockDim.x + threadIdx.x;
4
5      int l, k;
6      int j, i, j_id, i_id; // indexes in image
7      int Ix, Iy;           // gradient in XX and YY
```

```
    int R;                  // R metric
    int sumIx2, sumIy2, sumIxIy;

    if (id < size)
    {
        if (kernel_type == 0)
        {
            h_odata[id] = h_idata[id] / 4;  // Fade Image
        }

        else
        {
            i_id = id / w;                  // row, height
            j_id = id - (i_id * w);         // column

            i = ws + 1 + i_id;
            j = ws + 1 + j_id;

            sumIx2 = 0;
            sumIy2 = 0;
            sumIxIy = 0;

            for (k = -ws; k <= ws; k++) // height window
            {
                for (l = -ws; l <= ws; l++) // width window
                {
                    if ((i >= ws + 1) && (i < h - ws - 1) && (j >= ws + 1) && (j < w - ws - 1)
    )
                    {
                        Ix = ((int)h_idata[(i + k - 1) * w + j + l] - (int)h_idata[(i + k + 1)
     * w + j + l]) / 32;
                        Iy = ((int)h_idata[(i + k) * w + j + l - 1] - (int)h_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)
            {
                h_odata[i * w + j] = MAX_BRIGHTNESS;
            }
        }
    }
}
```

Listing 6: Harris Detector, Cuda Global memory, C/C++, implementação na função *reduce1*

Código disponível no(s) ficheiro(s):

- *harrisDetectorCuda.cu*

3.1.2 Análise de Resultados

As imagens processadas são idênticas às de referência. Na tabela seguinte é possível verificar os valores temporais obtidos.

Imagens	Tempo final Sequencial	Tempo final Paralelização	Eficiência %
chess.pgm	1.074656	0.374624	65.14%
chessBig.pgm	55.988224	7.223936	87.10%
house.pgm	2.060288	0.51072	75.21%
chessRotate1.pgm	1.032192	0.362368	64.89%
chessL.pgm	1.662976	0.485728	70.79%

Table II: CUDA, Global Memory - Tabela tempos finais.

3.2 Shared Memory

3.2.1 Metodologia

A implementação elaborada para CUDA em *shared memory*, Fig. 7, na função *harrisDetectorDevice* é parecida à implementação, referida anteriormente da *global memory*, apenas existe a omissão da segunda invocação do kernel e da variável *kernel_type*. desta forma o kernel apenas será chamado uma vez e, realizará o *fade* da imagem e o cálculo dos cantos na mesma evocação.

As diferenças são notáveis na função *reduce1*, Fig8. Nesta é visível a criação de um array horizontal (de uma dimensão) *sdata* de tamanho 33 para alojar o array input da imagem (*h_idata*), na linha 16 está expressa realização da cópia. Na linha 53 é possível a cópia da imagem processada para o array de output, neste caso cada *thread* realiza a cópia do seu pixel.

```
1     int size = h * w;
2     int memsize = size * sizeof(pixel_t);
3     int threadsPerBlock = 32;
4     int blocksPerGrid = size / threadsPerBlock;
5
6     // allocate host memory
7     pixel_t *devPtrh_idata;
8     pixel_t *devPtrh_odata;
9
10    // Allocate device memory
11    cudaMalloc((void **)&devPtrh_idata, memsize);
12    cudaMalloc((void **)&devPtrh_odata, memsize);
13    //cudaMalloc((void **)&devPtrsize, memsize_size_arr);
14
15    // Copy data (data to process) from host to device (from CPU to GPU)
16    cudaMemcpy(devPtrh_idata, h_idata, memsize, cudaMemcpyHostToDevice);
17    cudaMemcpy(devPtrh_odata, h_odata, memsize, cudaMemcpyHostToDevice);
18
19    // Call kernel to Fade image
20    // __global__ functions are called:  Func <<< dim grid, dim block >>> (parameter);
21    dim3 dimGrid(blocksPerGrid, 1, 1);
22    dim3 dimBlock(threadsPerBlock, 1, 1);
23
24    // Call kernel to calculate corners
25    // __global__ functions are called:  Func <<< dim grid, dim block >>> (parameter);
26    reduce1<<<dimGrid, dimBlock>>>(devPtrh_idata, devPtrh_odata, ws, w, threshold, h, size);
27
28    // Copy data from device (results) back to host
29    cudaMemcpy(h_odata, devPtrh_odata, memsize, cudaMemcpyDeviceToHost);
30
31    // Free device memory
32    cudaFree(devPtrh_idata);
33    cudaFree(devPtrh_odata);
```

Listing 7: Harris Detector, Cuda Shared memory, C/C++, implementação na função *harrisDetectorDevice*

```
1  __global__ void reduce1(pixel_t *h_idata, pixel_t *h_odata, int ws, int w, int threshold, int
       h, int size)
2  {
3      __shared__ pixel_t sdata[33];
4
5      int id = blockIdx.x * blockDim.x + threadIdx.x;
6      int tid = threadIdx.x;          // thread id de cada block
7
8      int l, k;
9      int j, i, j_id, i_id;           // indexes in image
10     int Ix, Iy;                     // gradient in XX and YY
11     int R;                          // R metric
12     int sumIx2, sumIy2, sumIxIy;
13
14     if (id < size)
15     {
16         sdata[tid] = h_idata[id];   // copy imagem array from global mem to shared
17
18         sdata[tid] = sdata[tid] / 4;// Fade Image
19
20         // Process Corners
21         i_id = id/w;                // row, height
22         j_id = id - (i_id*w);       // column
23
24         i = ws + 1 + i_id;
```

```
25          j = ws + 1 + j_id;
26
27          sumIx2 = 0;
28          sumIy2 = 0;
29          sumIxIy = 0;
30
31          if((i >= ws+1) && (i < h-ws-1) && (j >= ws+1) && (j < w-ws-1))
32          {
33              for (k = -ws; k <= ws; k++) // height window
34              {
35                  for (l = -ws; l <= ws; l++) // width window
36                  {
37                      Ix = ((int)h_idata[(i + k - 1) * w + j + l] - (int)h_idata[(i + k + 1) * w
      + j + l]) / 32;
38                      Iy = ((int)h_idata[(i + k) * w + j + l - 1] - (int)h_idata[(i + k) * w + j
      + l + 1]) / 32;
39                      sumIx2 += Ix * Ix;
40                      sumIy2 += Iy * Iy;
41                      sumIxIy += Ix * Iy;
42                  }
43              }
44          }
45
46          R = sumIx2 * sumIy2 - sumIxIy * sumIxIy - 0.05 * (sumIx2 + sumIy2) * (sumIx2 + sumIy2)
     ;
47          if (R > threshold)
48          {
49              sdata[tid] = MAX_BRIGHTNESS;
50          }
51
52          // Copy deteted corners to export array
53          h_odata[id] = sdata[tid];
54      }
55  }
```

Listing 8: Harris Detector, Cuda Shared memory, C/C++, implementação na função *reduce1*

Código disponível no(s) ficheiro(s):

- *harrisDetectorCudaShared.cu*

3.2.2 Análise de Resultados

Neste caso, a implementação não foi tão bem conseguida, pois o processamento da imagem não é realizado na memória partilhada, mas sim na global (linhas 37 a 41). De facto, seria uma versão verdadeiramente de memória partilhada caso somente este tipo de memória fosse utilizado durante o *o faded* e o processamento dos cantos. Apesar disto, as imagens processadas por esta implementação diferem das de referência por alguns pixels, havendo a deslocação, quase impercetível a olho nu, dos cantos assinalados. Esta situação é verificável em todas as imagens processadas. Desta forma, mesmo sendo pequenas variações, o script de teste *testDiffs* evidencia um valor elevado de diferenças.

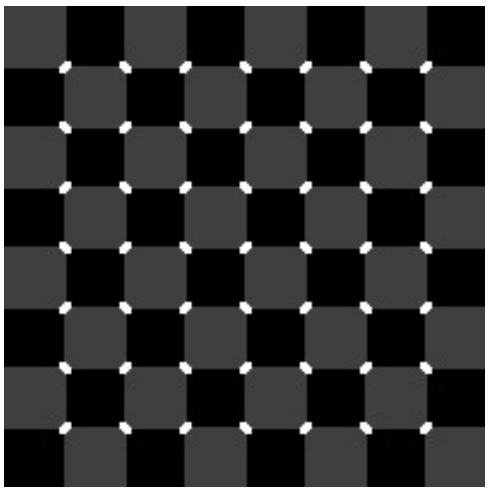
A fim de uma comparação de valores temporais com os da *global memory*, admitiu-se que as imagens processadas estariam razoavelmente corretas. Na tabela III é possivel verificar os valores.

Imagens	Tempo final Sequencial	Tempo final Paralelização	Eficiência %
chess.pgm	1.075008	0.328288	69.46%
chessBig.pgm	56.52512	5.524896	90.23%
house.pgm	2.134016	0.461536	78.37%
chessRotate1.pgm	1.132192	0.341152	69.87%
chessL.pgm	1.59568	0.3944	75.28%

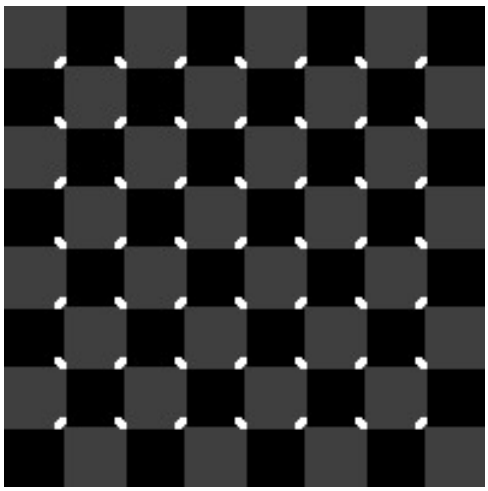
Table III: CUDA, Shared Memory - Tabela tempos finais.

3.3 Shared vs Global Memory

Comparando os valores das tabelas II e III, é possível verificar que existiu uma melhoria de eficiência generalizada. Apesar da implementação que não tira partido de todo o desempenho disponível na *shared memory*



(a) Imagem de referencia, *referenceCuda.pgm*



(b) Imagem processada, *resultCuda.pgm*

Fig. 6: CUDA, *shared memory*, comparação entre a imagem processada e de referência

(mencionado anteriormente), é notável a obtenção de melhores diferenças temporais, tendo, de certa forma, cumprido as expectativas iniciais.

4 OpenMP vs CUDA

Comparando os resultados obtidos do OpenMP num portátil mencionado e do CUDA numa CPU muito superior e com GPU mencionados, é possível verificar que a eficiência (percentagem de poupança de tempo) é, também, muito superior em CUDA para a imagem *chessBig.pgm*, podendo esta alcançar quase os 90% em relação aos 57% do OpenMP.

5 Testes de Validação

Foram realizados testes de eficiência e validação para OpenMP, CUDA global e CUDA shared. Esses testes encontram-se nas diretorias *testsOpenMP*, *testsCudaGlobal* e *testsCudaShared*, respetivamente.

5.1 OpenMP

De acordo com o output dos testes de validação, ou seja, da comparação entre imagens resultado e referências, as imagens são sempre idênticas. Esse output pode ser obtido executando *test_generateOpenMPfiles.py* e aceitando a opção de comparar as imagens. Um exemplo desse output encontra-se em *rawResultsOpenMP.txt*.

Código disponível na diretoria *testsOpenMP*, no(s) ficheiro(s):

- *test_generateOpenMPfiles.py*: gera vários scripts de OpenMP globais, um para cada versão de código diferente. Compila e executa esses scripts. Opcionalmente, compara imagens de resultados e referência gerados, executanto *testDiffs*, ficheiro gerado com *testDiffs.c*, um script desenvolvido pelo professor.
- *resultsOpenMP.py*: usa os resultados dos tempos finais de sequenciação e paralelização para calcular a eficiência. Guarda resultados numa tabela com marcação latex. A eficiência foi calculada com base na seguinte equação:

$$Eficiencia = \frac{(tempoSequencial - tempoParalelizacao) \times tempoParalelizacao}{100} \quad (1)$$

- *test_harrisDetectorOpenMP_scheduleguided8.c*: script gerado com *tests_generateOpenMPfiles.py*.
- *test_harrisDetectorOpenMP_scheduleguided2.c*: script gerado com *tests_generateOpenMPfiles.py*.
- *test_harrisDetectorOpenMP_scheduleguided30.c*: script gerado com *tests_generateOpenMPfiles.py*.

- *test_harrisDetectorOpenMP_schedulestatic8.c*: script gerado com *tests_generateOpenMPfiles.py*.
- *test_harrisDetectorOpenMP_scheduleguided8collapse2.c*: script gerado com *tests_generateOpenMPfiles.py*.
- *test_harrisDetectorOpenMP_scheduleguided2collapse2.c*: script gerado com *tests_generateOpenMPfiles.py*.
- *test_harrisDetectorOpenMP_scheduleguided30collapse2.c*: script gerado com *tests_generateOpenMPfiles.py*.
- *test_harrisDetectorOpenMP_schedulestatic8collapse2.c*: script gerado com *tests_generateOpenMPfiles.py*.
- *test_harrisDetectorOpenMP_scheduledynamic8collapse2.c*: script gerado com *tests_generateOpenMPfiles.py*.

Para executar os testes OpenMP, deve-se executar o script *test_generateOpenMPfiles.py*. Dependendo da preferência do utilizador, pode-se comparar as imagens geradas ou não. A seguir, para obter a percentagem de eficiência de cada teste, deve-se copiar manualmente o output do script anteriormente executado para *rawResultsOpenMP.txt* e executar *resultsOpenMP.py*.

5.2 CUDA Global

De acordo com o output dos testes de validação, as imagens são, em semelhança aos testes de OpenMP, sempre idênticas. Esse output pode ser obtido executando *tests_generateCudaScripts.py*, no servidor banana, e aceitando a opção de comparar as imagens. Um exemplo desse output encontra-se em *rawResultsCuda.txt*.

Código disponível na diretoria *testsCudaGlobal*, no(s) ficheiro(s):

- *tests_generateCudaScripts.py*: gera vários scripts de cuda globais, um para cada imagem. Compila e executa esses scripts. Opcionalmente, compara imagens de resultados e referência gerados, executanto *testDiffs*, ficheiro gerado com *testDiffs.c*, um script desenvolvido pelo professor.
- *resultsCuda.py*: usa os resultados dos tempos finais de sequenciação e paralelização para calcular a eficiência. Guarda resultados numa tabela com marcação latex. A eficiência foi calculada com base na equação (1).
- *test_harrisDetectorCuda_chess.cu*: script gerado com *tests_generateCudaScripts.py*.
- *test_harrisDetectorCuda_chessBig.cu*: script gerado com *tests_generateCudaScripts.py*.
- *test_harrisDetectorCuda_chessL.cu*: script gerado com *tests_generateCudaScripts.py*.
- *test_harrisDetectorCuda_chessRotate1.cu*: script gerado com *tests_generateCudaScripts.py*.
- test_harrisDetectorCuda_house.cu: script gerado com *tests_generateCudaScripts.py*.

Para executar os testes CUDA global, deve-se executar o script *tests_generateCudaScripts.py* no servidor banana. Dependendo da preferência do utilizador, pode-se comparar as imagens geradas ou não. A seguir, para obter a percentagem de eficiência de cada teste, no servidor banana ou localmente, copia-se manualmente o output do script para *rawResultsCuda.txt* e executa-se *resultsCuda.py*.

5.3 CUDA Shared

De acordo com o output dos testes de validação, as imagens são diferentes. Ao comparar as imagem resultado e de referência, é possível observar um deslocamento dos "traços brancos" detetados nas imagens. Esse output pode ser obtido executando *tests_generateCudaSharedScripts.py*, no servidor banana, e aceitando a opção de comparar as imagens. Um exemplo desse output encontra-se em *rawResultsCudaShared.txt*.

Código disponível na diretoria *testsCudaShared*, no(s) ficheiro(s):

- *tests_generateCudaScripts.py*: gera vários scripts de cuda globais, um para cada imagem. Compila e executa esses scripts. Opcionalmente, compara imagens de resultados e referência gerados, executanto *testDiffs*, ficheiro gerado com *testDiffs.c*, um script desenvolvido pelo professor.
- *resultsCuda.py*: usa os resultados dos tempos finais de sequenciação e paralelização para calcular a eficiência. Guarda resultados numa tabela com marcação latex. A eficiência foi calculada com base na equação (1).
- *test_harrisDetectorCudaShared_chess.cu*: script gerado com *tests_generateCudaSharedScripts.py*.

- *test_harrisDetectorCudaShared_chessBig.cu*: script gerado com *tests_generateCudaSharedScripts.py*.
- *test_harrisDetectorCudaShared_chessL.cu*: script gerado com *tests_generateCudaSharedScripts.py*.
- *test_harrisDetectorCudaShared_chessRotate1.cu*: script gerado com *tests_generateCudaSharedScripts.py*.
- test_harrisDetectorCudaShared_house.cu: script gerado com *tests_generateCudaSharedScripts.py*.

Para executar os testes CUDA shared, deve-se executar o script *tests_generateCudaSharedScripts.py* no servidor banana. Dependendo da preferência do utilizador, pode-se comparar as imagens geradas ou não. A seguir, para obter a percentagem de eficiência de cada teste, no servidor banana ou localmente, copia-se manualmente o output do script para *rawResultsCudaShared.txt* e executa-se *resultsCudaShared.py*.

6 Conclusão

O presente projeto possibilitou o estudo prático de aplicação de várias implementações e tipos de paralelização dois tipos de hardware, CPU e GPU, apesar disso, infelizmente não nos foi possível executar todas as implementações aconselhadas. Foi possível avaliar os resultados obtidos e interpretar os mesmos de acordo com a teoria lecionada nas aulas e mencionada e expandir o conhecimento a novas implementações existentes.

É interessante e motivador o estudo de uma área tão importante e prática quando a da Computação Paralela, é nela que estão assentes diversos projetos de investigação de elevada dimensão, por exemplo o SETI@Home, Folding@Home, etc.

Esperemos que a tecnologia continue a melhorar e, em comunidade, contribuir com ainda mais poder de processamento para inovar e melhorar o nosso dia a dia.

References

[1] https://en.wikipedia.org/wiki/LEO_(computer)

[2] https://pt.wikipedia.org/wiki/Multitarefa

[3] https://gist.github.com/mando7/d2c4464a5991f5787e9752e5950608fb

[4] https://www.amd.com/pt/processors/epyc-7003-series

[5] https://medium.com/data-breach/introduction-to-harris-corner-detector-32a88850b3f6

[6] https://en.wikipedia.org/wiki/OpenMP

[7] http://jakascorner.com/blog/2016/06/omp-for-scheduling.html

[8] https://nanxiao.gitbooks.io/openmp-little-book/content/posts/collapse-clause.html

[9] https://stackoverflow.com/questions/13357065/how-does-openmp-handle-nested-loops

[10] https://developer.nvidia.com/blog/using-shared-memory-cuda-cc/

[11] https://en.wikipedia.org/wiki/CUDA

[12] http://cuda.ce.rit.edu/cuda_overview/cuda_overview.htm

[13] https://www.tutorialspoint.com/cuda/cuda_memories.htm

[14] Powerpoints do primeiro Módulo de CP, Nuno Lau, Rui Costa, 2021-22