Capítulo 4

Computação heterogénia na reamostragem e redimensionamento de vídeo

~~Solução Proposta~~

Neste capítulo é descrita e detalhada a proposta de solução para o problema de aplicação do processo de reamostragem e redimensionamento de vídeo sem compressão segundo o modelo de cor YUV ~~atendendo o menor tempo possível de execução~~. ~~Em seguimento da~~ descrição da solução proposta são abordadas as fases que constituem a mesma e as respetivas otimizações implementadas. Por último, são discutidos os detalhes e as decisões de implementação tomadas, assim como as respetivas justificações.

4.1 Descrição

As imagens do modelo de cor YUV são caraterizadas por múltiplos formatos de pixeis, vários tipos de subamostragem de crominâncias e diferentes valores de profundidade de cor, como já referido na secção 3.4. Estes diferentes aspetos definem múltiplas combinações de caraterísticas de representação de imagens. Por esta razão, a solução deste trabalho deve considerar uma implementação que lida com as várias combinações de caraterísticas possíveis.

De modo a normalizar a lógica da solução deste trabalho foi apenas considerada a aplicação do processo de reamostragem e redimensionamento de imagens segundo o formato de pixeis planares. Caso a imagem a ser processada não apresente este tipo de formato, a última será convertida para um formato de pixeis planar que assegure os mesmos valores de profundidade de cor e o tipo de subamostragem de crominâncias quer da imagem original quer da imagem reamostrada. Então, a solução ~~foi~~ dividida em duas fases distintas: a fase de conversão de formatos de pixeis de frames e a fase de reamostragem e redimensionamento das imagens.

A lógica da solução implementada pode ser observada pelo diagrama de atividade da figura 4.1. Quando o processamento da operação de reamostragem e redimensionamento é aplicado a uma imagem sem indicação de alteração das suas dimensões, a última sofre apenas a operação da fase de conversão de formatos de pixeis: do formato inicial para o formato final desejado.

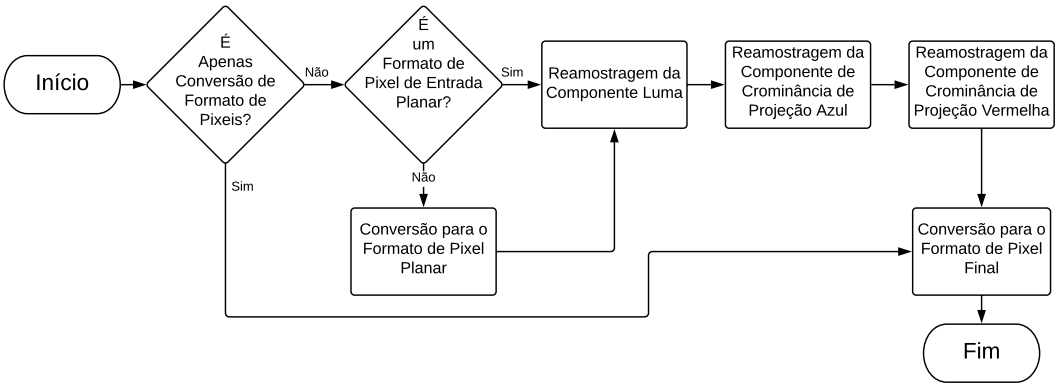

Figura 4.1: Diagrama de atividade da solução para o processamento de uma frame.

No caso em que exista a indicação da alteração das dimensões da imagem, o processamento é constituído pela sequência das seguintes três operações: a operação de conversão do formato de pixeis do formato de pixel da imagem original para um formato do tipo planar; de seguida, cada componente de cor (plano) da imagem é reamostrada e redimensionada individualmente recorrendo aos filtros de reconstrução; por último, a imagem reamostrada e redimensionada sofre novamente a operação de conversão do tipo de formato de pixeis, do formato planar temporário para o formato final desejado.

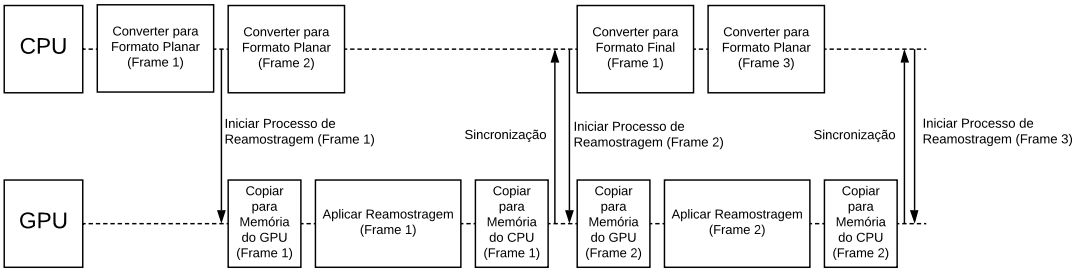

Figura 4.2: Diagrama de organização de tarefas de processamento das frames de um vídeo.

A figura ~~anterior~~ representa ~~a esquemática da~~ organização da atividade da solução desenvolvida no processamento das primeiras três frames de um vídeo, sendo que o restante processamento é uma replicação do padrão de operações apresentado (ver também o apêndice A.1).

De modo a tirar o máximo partido das capacidades computacionais do CPU e do GPU, ambas unidades de processamento executam diferentes operações em simultâneo, havendo apenas sincronização de ambas execuções entre operações de reamostragem e redimensionamento de frames de modo a que o processamento realizado não seja descoordenado e ineficiente. A solução foi implementada de forma a que a operação de conversão de formato de pixeis fosse realizada apenas pelo CPU e a operação de reamostragem e redimensionamento pelo GPU.

Considerando uma frame F_n de índice n de um vídeo, as duas unidades de processamento

executam as fases da solução referidas acima da seguinte forma: enquanto a operação de reamostragem e redimensionamento da frame F_n é levada a cabo pelo GPU, o CPU realiza as operações de conversão do tipo de formatos de pixeis do formato planar intermédio para o formato final e do formato de pixeis inicial para o formato planar intermédio das frames F_{n-1} e F_{n+1}, respetivamente.

Esta divisão de processamento em fases permite uma utilização dos recursos computacionais das unidades de processamento próxima do valor máximo. A figura 4.3 apresenta a percentagem do tempo de execução de cada uma das fases referidas no processamento de um vídeo segundo a implementação desta solução.

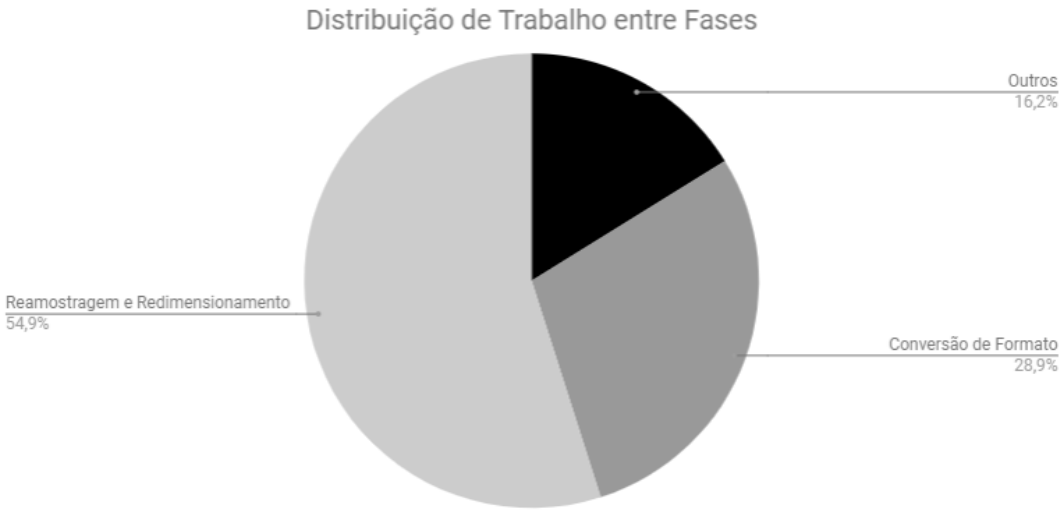

Figura 4.3: Distribuição de trabalho das fases de processamento.

A fase de reamostragem e redimensionamento de uma frame, executada no GPU, é o fator limitante de toda esta operação visto que é a operação mais morosa de toda a execução. Por essa razão, a utilização do CPU não é máxima, pois a fase de conversão do tipo de formatos de pixeis das frames F_{n-1} e F_{n+1} apresenta um tempo de execução menor que a fase de reamostragem e redimensionamento da frame F_n. Este acontecimento leva a que o CPU tenha períodos de inatividade enquanto aguarda os seguintes dados a serem processado e, por consequência, a utilização das suas capacidade computacionais não são levadas ao máximo.

De modo a evitar esta peculiaridade e ~~a~~ maximizar a utilização do CPU, o último poderia continuar a sua execução e processar as restantes frames do vídeo, ao invés de bloquear e esperar pelo término da fase do GPU. Porém, este tipo de implementação é impossível visto que a solução tem como objetivo a aplicação do processo de pós-produção - reamostragem e redimensionamento - de vídeo *live*, que se encontra em captura, e por essa razão o processamento das frames seguintes pode não ser possível pois podem ainda não terem sido capturadas.

4.2 Operação de Conversão de Formatos de Pixeis

A operação de conversão do tipo de formatos de pixeis de uma imagem é realizada exclusivamente recorrendo às capacidades computacionais do CPU utilizando a ferramenta OpenMP para a sua implementação. A operação de conversão do formato de pixeis de uma imagem é fundamentalmente um problema de reorganização dos valores das componentes de cor dos seus pixeis. Por essa razão, não se justifica a utilização das capacidades computacionais do GPU, visto que a última está, exclusivamente, otimizada para a execução de problemas de cariz aritmético [18].

As limitações da aplicação desta operação devem-se ao elevado número de acessos a memória necessários de serem realizados para efetuar as operações de leitura e escrita dos valores de intensidade de cor de cada componente de um pixel. Assim, a chave para a otimização deste problema encontra-se na utilização de técnicas que implementem eficientemente os acessos a memória.

Os acessos aleatórios a memória são custosos devido à latência de transferência dos dados armazenados na memória para os registos do processador caso os dados não se encontrem já armazenados no sistema de cache, os designados *page faults* [35]. Para minimizar a ocorrência de *page faults* é necessário ter em conta o comportamento do sistema de memória cache. Isto é, o número de ocorrências de *page faults* é reduzido caso os acessos a memória sejam feitos a regiões próximas de acessos realizados anteriormente de modo a ter uma maior probabilidade de encontrar os dados no sistema de cache [36].

Tendo em conta que a realização da operação de conversão do tipo de formatos pixeis de uma frame é executada por vários *threads*, implementados com a ferramenta OpenMP, é necessário assegurar uma divisão equilibrada dos recursos computacionais para efetuar a operação atendendo a um acesso eficiente a memória.

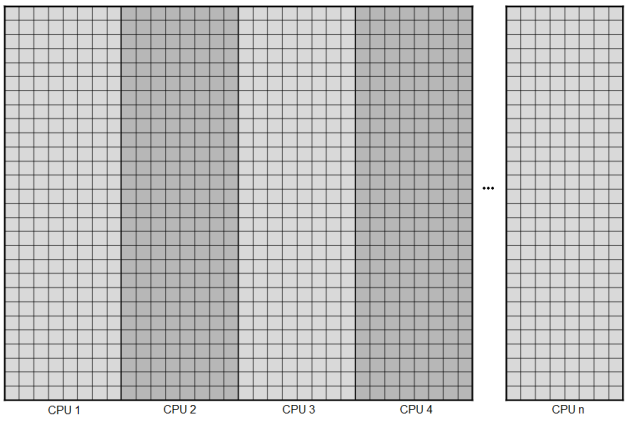

Figura 4.4: Divisão do processamento de uma frame em regiões.

A figura 4.4 estrutura a divisão de uma frame de um vídeo em regiões a serem processadas por diferentes *threads*. Cada quadrícula representa os pixeis de uma imagem e cada região sombreada os dados de uma imagem a serem atribuídos a cada *thread* dos *n* cores do CPU. Esta divisão permite que os acessos a memória conduzidos por cada um dos *threads* aceda a regiões de memória

próximas das acedidas pelos restantes *threads*, tirando maior partido do sistema de memória cache e melhorando a eficiência de acessos a memória, devido à redução de *page faults*.

O seguinte excerto de código exemplifica a implementação da operação de conversão do tipo de formato de pixeis com a ferramenta OpenMP praticada na solução desenvolvida. O ciclo *for* exterior itera o processamento de todas as linhas da imagem, enquanto o ciclo interior itera todas as colunas da mesma.

A diretiva *pragma omp parallel* declara um bloco de código paralelo a ser executado em múltiplos *threads* do CPU como uma tarefa, enquanto a diretiva *pragma omp for schedule(static)*, imediatamente antes do ciclo *for* interior, dá a indicação ao CPU de paralelização do ciclo por todos os *threads* da secção paralelizada.

A carga de trabalho do ciclo será dividida por todos os *threads* de acordo com um certo padrão especificado pela palavra-chave *schedule(static)*. Esta diretiva dá indicação ao CPU que a divisão dos dados será realizada estaticamente durante a compilação de modo a que cada *thread* seja responsável pelo processamento de blocos de dados de dimensão $\frac{IMAGE\_WIDTH}{THREAD\_COUNT}$, sendo a variável $IMAGE\_WIDTH$ o valor da dimensão de largura da imagem e a variável $THREAD\_COUNT$ o número de threads que constituem a secção de código paralelizada.

```
1  #pragma omp parallel
2  {
3      // Itera por todas as linhas da imagem
4      for(int vIndex = 0; vIndex < IMAGE_HEIGHT; vIndex++){
5          // Paraleliza o processamento das colunas da imagem
6          #pragma omp for schedule(static)
7          for(int hIndex = 0; hIndex < IMAGE_WIDTH; hIndex++){
8              // Converte o formato de pixel do bloco de pixeis da linha vIndex
9              convertPixelFormatOfImageBlock(vIndex, hIndex);
10         }
11     }
12 }
```

Nos casos em que a operação de reamostragem e redimensionamento indica alterações das dimensões da imagem, a operação de conversão do tipo de formatos de pixeis do formato original para o formato planar intermédio é realizado de modo a reduzir o número de operações lógicas do processo de reamostragem. A redução do número de operações deve-se à organização dos dados dispostos por planos ao invés de necessitar de operações adicionais para organizar os valores de componentes de cor entrelaçados.

Adicionalmente, a quantidade de dados a serem processados na operação de reamostragem e redimensionamento são reduzidos pela conversão de formato de pixeis, pois o formato planar intermédio terá o mesmo tipo de subamostragem de crominâncias que o tipo de subamostragem com menos valores de componentes de crominâncias da imagem original ou da imagem reamostrada. Assim, como exemplo, o processo de reamostragem e redimensionamento de um vídeo, caraterizado pelo tipo de subamostragem de crominâncias 4:2:2 para uma subamostragem do tipo

4:2:0, irá ser realizado com um formato planar intermédio com o tipo de subamostragem de crominâncias 4:2:0, reduzindo efetivamente o número de valores de intensidade de componentes de cor a serem processados em 25%.

A tabela 4.1 mostra o tipo de subamostragem de crominâncias do formato de pixeis planar intermédio pelo qual é realizada a operação de reamostragem e redimensionamento, dependendo do tipo de formato de pixeis de entrada, da imagem original, e de saída, da imagem reamostrada.

Tabela 4.1: Tipo de subamostragem de crominâncias do formato de pixeis planar intermédio

	Formato de Pixeis da Imagem Reamostrada				
Formato de Pixeis da Imagem Original	UYVY	YUV422p	YUV420p	NV12	V210
UYVY	4:2:2	4:2:2	4:2:0	4:2:0	4:2:2
YUV422p	4:2:2	4:2:2	4:2:0	4:2:0	4:2:2
YUV420p	4:2:0	4:2:0	4:2:0	4:2:0	4:2:0
NV12	4:2:0	4:2:0	4:2:0	4:2:0	4:2:0
V210	4:2:2	4:2:2	4:2:0	4:2:0	4:2:2

4.3 Operação de Reamostragem e Redimensionamento

A operação de reamostragem e redimensionamento é realizada exclusivamente recorrendo às capacidades computacionais do GPU utilizando a plataforma de desenvolvimento CUDA para a sua implementação. como referido na secção 3.3.1, a operação de reamostragem e redimensionamento de uma imagem é fundamentalmente a aplicação do cálculo de interpolação recorrendo aos filtros de reconstrução. Esta operação é conseguida pela aplicação de cálculos aritméticos sobre os dados das componentes de cor de uma imagem e, por essa razão, a natureza desta operação torna a utilização do GPU extremamente eficaz para a resolução deste problema [19].

A alocação de memória de uma unidade de processamento gráfico é uma operação síncrona com o CPU, bloqueando o último até o término dessa atividade. Como as caraterísticas da operação de reamostragem e redimensionamento de um vídeo são semelhantes para todas as suas frames, e tendo em conta que durante esta fase apenas uma frame é processada de cada vez, o processo de alocação de memória pode ser realizado apenas uma vez no início de toda esta operação e libertada no seu final reservando uma região de memória de dimensões igual às componentes de cor de uma frame do vídeo original e do vídeo reamostrado. A cada iteração do processamento de um vídeo, a memória do GPU recebe os dados das componentes de cor de uma frame sobrepondo os valores da frame anterior.

As limitações da aplicação desta operação são introduzidas pelas operações inevitáveis de transferência dos dados das componentes de cor das frame entre o dispositivo de memória da máquina e da unidade de processamento gráfico, e o elevado tempo de resposta a acessos de memória. Estas limitações são provocadas, respetivamente, devido ao GPU apenas conseguir operar sobre dados que se encontrem em algum dos seus níveis de memória e à natureza de acessos não coalescidos a memória desta operação, como será explicado de seguida.

Uma das caraterísticas dos filtros de reconstrução utilizados na operação de reamostragem e redimensionamento é o seu valor de apoio de pixeis. O valor de apoio de pixeis indica o número de pixeis da imagem original mais próximos da posição do pixel da imagem reamostrada que são tidos em conta no cálculo do seu valor de intensidade de cor.

O valor de intensidade de cor de cada pixel da imagem reamostrada é obtido a partir do cálculo de interpolação por cada *thread* do GPU. Cada um dos *threads* necessita de realizar vários acessos a memória para recuperar os valores de intensidade de cor dos pixeis originais.

Assim, um *thread* não acede de forma coalescida a memória do GPU. Porém, como cada *thread* acede aos valores de intensidade de cor dos pixeis da imagem original mais próximos da posição do pixel da imagem reamostrada, é possível tirar partido do sistema de cache da memória de textura do GPU por priorizar a localidade espacial dos dados (a proximidade das posições acedidas da memória) e assim reduzir a latência de resposta a acessos de memória por parte de cada *thread* [14], como descrito na secção 2.3.1.4.

4.3.1 Mecanismos de Transferência de Dados do GPU

A plataforma de desenvolvimento CUDA fornece diferentes mecanismos de transferência que permitem a cópia de dados a partir da memória da máquina para os níveis de memória das unidades de processamento gráfico. Esses métodos são os seguintes:

- Cópia de memória sincronizada - Este mecanismo permite reservar uma região de um dos níveis de memória do GPU onde serão armazenados os dados transferidos da memória da máquina. Esta transferência é realizada sincronamente com o CPU, bloqueando-o até o término da atividade.

- Memória *pinned* - O mecanismo de memória *pinned* é utilizado pelo GPU para executar operações sobre dados sem que haja uma cópia explicita dos mesmos da memória da máquina para a memória da unidade de processamento gráfico. Este mecanismo é realizado através do procedimento DMA do GPU, ou *direct memory access*. O último é um procedimento que permite o acesso a memória sem que haja bloqueio de atividade da unidade de processamento que o invocou. O mecanismo de memória *pienned* permite a execução direta do GPU sobre dados armazenados na memória da máquina.

- Cópia de memória não paginada - Este tipo de transferência de dados é conseguido através da declaração como não paginável de uma região de memória da máquina. Esta declaração indica ao sistema operativo que a região não deve ser movida ou copiada para um ficheiro de paginação de memória. Ao declarar uma região de memória da máquina como não paginável, o GPU consegue realizar uma cópia dos seus dados assincronamente, não bloqueando a atividade quer do CPU quer da unidade de processamento gráfico.

Dos três mecanismos de transferência de dados entre a memória da máquina e do GPU, o mecanismo de memória não paginada é o que apresenta melhor desempenho e escalabilidade para

soluções de processamento paralelo de dados [37], pelo que foi o mecanismo implementado na solução deste trabalho.

4.3.2 Divisão de ~~Trabalho~~

A plataforma de desenvolvimento CUDA estabelece o conceito de sequência de operações com a declaração de uma classe designada de *stream*. As instâncias desta classe permitem a paralelização de diferentes tarefas como a transferência de dados para a memória do GPU a partir de memória não paginável, a operação reversa e a execução de uma operação de processamento sobre os dados. Uma unidade de processamento gráfico tem a capacidade de executar as referidas tarefas em simultâneo desde que as últimas sejam de diferentes tipos e pertencentes a instâncias diferentes da classe *stream*.

Figura 4.5: Divisão de tarefas por *streams* no GPU.

A figura ~~anterior~~ reflete a divisão dos diferentes tipos de tarefas realizadas concorrentemente pelo GPU durante a operação de reamostragem e redimensionamento desta solução. Na figura 4.5, cada linha carateriza uma instância da classe *stream*, os blocos coloridos *Y*, *U* e *V* representam as operações de reamostragem e redimensionamento das três componentes de cor de uma frame, e os blocos *A1*, *A2*, *A3*, *B1*, *B2* e *B3* representam as transferências de dados entre o dispositivo de memória da máquina e o GPU.

Com a utilização de instâncias da classe *stream* foi permitido esconder a latência causada pelas transferências de dados, sobrepondo essas atividades com a execução das operações de reamostragem e redimensionamento à medida que os recursos são disponibilizados pelo GPU.

~~O seguinte excerto de código~~ implementa a divisão de dados por tarefas e a sobreposição da sua execução encontrada na figura 4.5. Nesta implementação o problema é divido no processamento através da operação de reamostragem e redimensionamento de cada componente de cor de cada frame de um vídeo. As *streams* são criadas tendo em vista o processamento em simultâneo das diferentes componentes. Esta implementação permite que o CPU sobreponha outras execuções que tenha a realizar, em específico a operação de conversão do formato de pixeis de outras frames, enquanto a operação de reamostragem e redimensionamento é executado assincronamente em cada instância da classe *stream*.

```
1  // Aloca memoria na memoria de textura do GPU
2  cudaArray* Y_COMPONENT_GPU_ARRAY, * U_COMPONENT_GPU_ARRAY, * V_COMPONENT_GPU_ARRAY;
3  cudaMallocArray(&Y_COMPONENT_GPU_ARRAY, &TRANSFER_CHANNEL, WIDTH, HEIGHT);
4  cudaMallocArray(&U_COMPONENT_GPU_ARRAY, &TRANSFER_CHANNEL, CHROMA_WIDTH,
       CHROMA_HEIGHT);
5  cudaMallocArray(&V_COMPONENT_GPU_ARRAY, &TRANSFER_CHANNEL, CHROMA_WIDTH,
       CHROMA_HEIGHT);
6
7  // Aloca memoria no GPU atraves do mecanismo de memoria nao paginavel
8  uint8_t* Y_RESAMPLED_GPU_ARRAY, * U_RESAMPLED_GPU_ARRAY, * V_RESAMPLED_GPU_ARRAY;
9  cudaMallocHost((void **) Y_RESAMPLED_GPU_ARRAY, Y_RESAMPLED_COMPONENT_SIZE);
10 cudaMallocHost((void **) U_RESAMPLED_GPU_ARRAY, U_RESAMPLED_COMPONENT_SIZE);
11 cudaMallocHost((void **) V_RESAMPLED_GPU_ARRAY, V_RESAMPLED_COMPONENT_SIZE);
12
13 // Instanciacao das diferentes streams
14 cudaStream_t streamY, streamU, streamV;
15 cudaStreamCreate(&streamY);
16 cudaStreamCreate(&streamU);
17 cudaStreamCreate(&streamV);
18
19 // Copia assincronamente os dados para a memoria de textura do GPU
20 cudaMemcpyToArrayAsync(Y_COMPONENT_GPU_ARRAY, 0, 0, Y_COMPONENT_CPU_ARRAY,
       Y_COMPONENT_SIZE, cudaMemcpyHostToDevice, streamY);
21 cudaMemcpyToArrayAsync(U_COMPONENT_GPU_ARRAY, 0, 0, U_COMPONENT_CPU_ARRAY,
       U_COMPONENT_SIZE, cudaMemcpyHostToDevice, streamU);
22 cudaMemcpyToArrayAsync(V_COMPONENT_GPU_ARRAY, 0, 0, V_COMPONENT_CPU_ARRAY,
       V_COMPONENT_SIZE, cudaMemcpyHostToDevice, streamV);
23
24 // Operacao de reamostragem e redimensionamento das componentes de cor
25 resampleOperation<<<KERNEL_lAUNCH_PARAMETERS, streamY>>>(SOURCE_WIDTH,
       SOURCE_HEIGHT, TARGET_WIDTH, TARGET_HEIGHT, Y_COMPONENT_GPU_ARRAY,
       Y_RESAMPLED_GPU_ARRAY);
26 resampleOperation<<<KERNEL_lAUNCH_PARAMETERS, streamU>>>(SOURCE_WIDTH,
       SOURCE_HEIGHT, TARGET_WIDTH, TARGET_HEIGHT, U_COMPONENT_GPU_ARRAY,
       U_RESAMPLED_GPU_ARRAY);
27 resampleOperation<<<KERNEL_lAUNCH_PARAMETERS, streamV>>>(SOURCE_WIDTH,
       SOURCE_HEIGHT, TARGET_WIDTH, TARGET_HEIGHT, V_COMPONENT_GPU_ARRAY,
       V_RESAMPLED_GPU_ARRAY);
28
29 // Copia assincronamente os dados da memoria global do GPU
30 cudaMemcpyAsync(Y_COMPONENT_CPU_ARRAY, Y_RESAMPLED_GPU_ARRAY,
       Y_RESAMPLED_COMPONENT_SIZE, cudaMemcpyDeviceToHost, streamY);
31 cudaMemcpyAsync(U_COMPONENT_CPU_ARRAY, U_RESAMPLED_GPU_ARRAY,
       U_RESAMPLED_COMPONENT_SIZE, cudaMemcpyDeviceToHost, streamU);
32 cudaMemcpyAsync(V_COMPONENT_CPU_ARRAY, V_RESAMPLED_GPU_ARRAY,
       V_RESAMPLED_COMPONENT_SIZE, cudaMemcpyDeviceToHost, streamV);
```

4.4 Comparação de *float* e *double*

A utilização das unidades de processamento gráfico na execução da operação de reamostragem e redimensionamento de frames de um vídeo restringe a precisão dos tipos de dados das variáveis utilizadas. Os GPUs têm uma maior afinidade com a utilização do tipo *float* de dados relativamente ao tipo *double*. Os GPUs manifestam um melhor tempo de execução de processamento de cálculos constituídos por operações com dados do tipo *float*, ou de precisão singular, em comparação com o tempo de execução de cálculos constituídos por dados do tipo *double*, ou de precisão dupla [38]. Esta discrepância deve-se à arquitetura das unidades de processamento gráfico, pois estas unidades são constituídas por um maior número de componentes de processamento de operações de vírgula flutuante de precisão singular do que de precisão dupla [3].

Nesta secção será abordada a análise numérica da precisão associada à operação de reamostragem e redimensionamento utilizando variáveis do tipo *float* e do tipo *double*, comparando o impacto de cada um desses tipos na qualidade da imagem resultante.

4.4.1 Estrutura de Números de Vírgula Flutuante ~~*float* e *double*~~

Um número real é representado pela arquitetura de uma máquina segundo o standard IEEE 754, que especifica a arquitetura e aritmética computacional associada aos números de vírgula flutuante [39]. Segundo este standard, os tipos de dados *float* e *double* são representados pelos campos de sinal, expoente e mantissa.

Tabela 4.2: Tamanho em bits dos campos dos tipos de dados *float* e *double*

	Tamanho em bits	
Campo	Float	Double
Sinal	1	1
Expoente	8	11
Mantissa	23	52

A tabela anterior mostra o tamanho em bits dos campos dos tipos de dados *float* e *double* para a representação de números de vírgula flutuante, segundo o standard IEEE 754. O número de bits dedicados ao expoente determina o alcance de grandeza do valor representado pelo tipo de dados da variável, enquanto o número de bits da mantissa determina a precisão da representação do seu valor.

Considerando um tipo de dados de vírgula flutuante com E bits reservados à representação do expoente, o alcance de grandeza do seu valor é dado por $[-2^{E-1}+1, 2^{E-1}] : E \in \mathbb{N}$. Como se pode induzir pela formulação matemática do alcance de grandeza do expoente, um dos bits do expoente não é utilizado para a representação do valor de expoente de um número real, pois segundo o standard IEEE 754 esse bit é utilizado no tratamento de exceções como o número zero e o número NAN, ou *not a number*, que representa valores de vírgula flutuante não representáveis ou não definidos [39].

4.4.2 Erro de Arredondamento Associado aos Tipo de Dados *float* e *double*

Devido à limitação do número de bits reservados à representação da mantissa na arquitetura de uma máquina, existe um arredondamento do número real ao número de vírgula flutuante mais próximo, representável como *float* ou *double*.

Como exemplo, o número $(0.1)_{10}$ tem uma representação infinita em binário: $(1100(1100))_2$. De modo a poder representar este número como um número de vírgula flutuante, segundo o standard IEEE 754, é necessário restringi-lo ao número de bits da mantissa através de uma operação de arredondamento. O valor do arredondamento de um número real ao número de vírgula flutuante mais próximo representável pode ser entendido como o erro associado a essa conversão.

A função ULP, ou *Unit of Least Precision*, de um número real é o valor do bit menos significativo da mantissa na sua representação como um número de vírgula flutuante do standard IEEE 754. O valor da função ULP é o valor da diferença entre dois números de vírgula flutuante consecutivos. A função ULP tem um comportamento proporcionalmente direto à magnitude do valor do número real considerado [40].

Considera-se um número real x e os números de vírgula flutuante representáveis pelos tipos do standard IEEE 754 x_a e x_b mais próximos de x, de tal modo que $x_a < x < x_b$. Pela definição da função ULP, $x_b - x_a = ULP(x)$. Assim, se $\frac{ULP(x)}{2} < x_b - x$, então $x - x_a < \frac{ULP(x)}{2}$. Logo, é possível assumir que o erro máximo causado pelo arredondamento de um número real ao número de vírgula flutuante representável mais próximo, dos tipos especificados pelo standard, é no máximo $\frac{ULP(x)}{2}$.

De modo a generalizar o erro de arredondamento associado à representação de números reais como números de vírgula flutuante dos tipos *float* e *double*. Considera-se um tipo de representação genérica com E e P bits reservados, respetivamente, ao expoente e à mantissa. O erro associado ao arredondamento de um número real x é formulado segundo a seguinte equação:

$$Erro(x) = \frac{ULP(x)}{2} = \frac{max(2^{-149}, 2^{\lfloor log_2(|x|) - P + 1 \rfloor})}{2}, x \in [-2^{E-1}+1, 2^{E-1}] : \{x, E, P\} \in \mathbb{N} \quad (4.1)$$

O valor do erro associado ao arredondamento de um número real à representação do número de vírgula flutuante correspondente, segundo o standard IEEE 754, tem um comportamento proporcionalmente direto à grandeza do primeiro. Isto é, quanto maior for o número real a ser representado como *float* ou *double*, maior será o erro associado à operação de arredondamento da conversão, de tal modo que:

$$Erro(x) < Erro(y), \quad sex < y \quad (4.2)$$

Por isso, tendo em conta a profundidade de cor de oito bits dos formatos de pixeis do modelo de cor YUV considerados neste trabalho, o maior valor a ser convertido para um dos tipos de números de vírgula flutuante é igual a $2^8 - 1 = 255$. Então, os maiores valores de erros associados à operação de arredondamento da conversão para os tipos de dados *float* e *double* são, respetivamente:

$$Erro_{float}(255) = \frac{ULP_{float}(255)}{2} = 2^{-17}$$
$$Erro_{double}(255) = \frac{ULP_{double}(255)}{2} = 2^{-46} \tag{4.3}$$

4.4.3 Análise Numérica da Operação de Reamostragem e Redimensionamento

Como referido em 4.2 e em 4.3, apenas a operação de reamostragem e redimensionamento é realizada através de operações aritméticas, pois a operação de conversão do formato de pixeis de uma imagem envolve apenas operações lógicas e a reorganização da memória. Por essas razões, a análise numérica de toda a solução pode ser efetuada analisando numericamente a operação de reamostragem e redimensionamento.

Seguindo a formulação da operação de reamostragem de imagens como convolução, apresentada em 3.10, o valor de intensidade de cor de um pixel de uma imagem reamostrada é obtido através do somatório do produto dos filtros de reconstrução e os valores de intensidade de cor dos a^2 pixeis mais próximos da imagem original, sendo a o valor de apoio de pixeis do filtro de reconstrução.

Assim o erro de arredondamento associado à conversão de números reais aos tipos *float* e *double* de número de vírgula flutuante pode ser formulado como a seguinte equação, sendo Δ o valor de incerteza associado ao resultado da convolução:

$$\Delta I_t(x,y) = \sum_{i=\lfloor u \rfloor - a + 1}^{\lfloor u \rfloor + a} \sum_{j=\lfloor v \rfloor - a + 1}^{\lfloor v \rfloor + a} Erro(I_s(\lfloor u \rfloor, \lfloor v \rfloor)) \times \Delta k(u-i) \times \Delta k(v-j) \tag{4.4}$$

Como $I_s(x,y)$ é a função de mapeamento dos pixeis da imagem original, o seu resultado é o valor da intensidade de cor das componentes do respetivo pixel. Então, o racional apresentado em 4.4.2 pode ser aplicado no cálculo do valor de $Erro(I_s(\lfloor u \rfloor, \lfloor v \rfloor))$, isto é, considera-se o valor de intensidade de pixel que apresenta um maior erro de arredondamento:

$$Erro_{float}(I_s(\lfloor u \rfloor, \lfloor v \rfloor)) = Erro_{float}(255) = 2^{-17} \tag{4.5}$$

$$Erro_{double}(I_s(\lfloor u \rfloor, \lfloor v \rfloor)) = Erro_{double}(255) = 2^{-46} \tag{4.6}$$

4.4.3.1 Análise Numérica dos Filtros de Reconstrução

O parâmetro dos filtros de reconstrução é a distância entre a posição do pixel da imagem reamostrada no sistema de coordenadas da imagem original e as posições dos a^2 pixeis mais próximos na imagem original. Assim, o valor do parâmetro de $k(x)$ está compreendido entre os valores do intervalo $[-\frac{a}{2}, \frac{a}{2}]$. Como os filtros de reconstrução lidam com as distâncias entre os pixeis, o valor absoluto do seu parâmetro de entrada poderá tomar os valores do intervalo: $[0, \frac{a}{2}]$.

Com o filtro de reconstrução *nearest neighbor*, sendo $a = 2$ o valor de apoio de pixeis, o parâmetro de entrada pode tomar valores do intervalo $[0,1]$. O valor de incerteza dos resultados do filtro, devido à propagação dos erros de arredondamento associados à utilização dos tipos *float* e *double*, é formulado como:

$$\Delta k_{nn}(x) = \begin{cases} Erro(1) & x < 0.5 \\ Erro(0) & senão \end{cases} \tag{4.7}$$

Assim, o valor de incerteza dos resultados deste filtro, assumindo o pior cenário possível, é:

$$\Delta k_{nn}(x) = max(Erro(1), Erro(0)) = Erro(1) \tag{4.8}$$

Quanto ao filtro de reconstrução linear, o valor de $a = 2$ implica que o parâmetro de entrada toma valores do intervalo $[0,1]$. O valor de incerteza deste filtro de reconstrução é dado como:

$$\Delta k_{linear}(x) = \begin{cases} \sqrt{Erro(1)^2 + Erro(x)^2} & x < 1 \\ Erro(0) & senão \end{cases} \tag{4.9}$$

$$\Delta k_{linear}(x) = max(\sqrt{Erro(1)^2 + Erro(x)^2}, Erro(0)) = \sqrt{Erro(1)^2 + Erro(x)^2} \tag{4.10}$$

Por último, o filtro de reconstrução por *spline* de Mitchell e Netravali é caraterizado pelo valor de apoio de pixeis igual a $a = 4$. Este valor indica que o parâmetro de entrada do filtro de reconstrução toma valores do intervalo $[0,2]$. A seguinte equação formula a incerteza do valor resultante do filtro:

$$\Delta k_{spline}(x) = \begin{cases} \Delta(1.4x^3 - 2.4x^2 + 1) & x < 1 \\ \Delta(-0.6x^3 + 3x^2 - 4.8x + 2.4) & 1 \leq x < 2 \\ Erro(0) & senão \end{cases} \tag{4.11}$$

Os valores de incerteza de cada um dos ramos do filtro de reconstrução considerados na anterior equação são demonstradas de seguida através do racional teórico ~~por de trás~~ da análise de propagação de erros, como descrito ~~na literatura~~ [41].

$$\begin{aligned}
&\Delta(1.4x^3 - 2.4x^2 + 1) = \\
&(\Delta(1.4x^3 - 2.4x^2)^2 + Erro(1)^2)^{\frac{1}{2}} = \\
&(\Delta(1.4x^3)^2 + \Delta(-2.4x^2)^2 + Erro(1)^2)^{\frac{1}{2}} = \\
&((1.4 \times \Delta(x^3) + x^3 \times Erro(1.4))^2 + (-2.4 \times \Delta(x^2) + x^2 \times Erro(2.4))^2 + Erro(1)^2)^{\frac{1}{2}} = \\
&((\frac{4.2 \times Erro(x)}{x} + x^3 \times Erro(1.4))^2 + (\frac{4.8 \times Erro(x)}{x} + x^2 \times Erro(2.4))^2 + Erro(1)^2)^{\frac{1}{2}}
\end{aligned} \tag{4.12}$$

Similarmente, aplicando a análise no ramo complementar do mesmo filtro de reconstrução:

$$\begin{aligned}
\Delta(-0.6x^3 + 3x^2 - 4.8x + 2.4) =&((\frac{-1.8 \times Erro(x)}{x} + x^3 \times Erro(-0.6))^2 + \\
&(\frac{6 \times Erro(x)}{x} + x^2 \times Erro(3))^2 + \\
&(4.8 \times Erro(x) + x \times Erro(4.8))^2 + \\
&Erro(2.4)^2)^{\frac{1}{2}}
\end{aligned} \tag{4.13}$$

Como $\Delta(1.4x^3 - 2.4x^2 + 1) < \Delta(-0.6x^3 + 3x^2 - 4.8x + 2.4)$ pode-se concluir que:

$$\begin{aligned}
\Delta k_{spline}(x) =& max(\Delta(1.4x^3 - 2.4x^2 + 1), \\
&\Delta(-0.6x^3 + 3x^2 - 4.8x + 2.4), \\
&Erro(0)) = \\
&\Delta(-0.6x^3 + 3x^2 - 4.8x + 2.4)
\end{aligned} \tag{4.14}$$

Em suma, a incerteza do resultado de cada um dos filtros de reconstrução, devido à propagação de erros causados pelo arredondamento de números reais aos tipos de dados de números de vírgula flutuante *float* e *double*, são apresentadas na tabela ~~seguinte:~~

Tabela 4.3: Incerteza dos resultados dos filtros de reconstrução

	k_{nn}	k_{linear}	k_{spline}
Float	1.19×10^{-7}	1.68×10^{-7}	6.90×10^{-4}
Double	2.22×10^{-16}	3.14×10^{-16}	2.98×10^{-8}

4.4.3.2 Análise Numérica da Convolução

Como o cálculo do valor de intensidade de cor de um pixel da imagem original na operação de reamostragem durante a convolução é obtido pela a aplicação do filtro de reconstrução nas

duas dimensões da imagem, na horizontal e vertical, a incerteza associada ao peso obtido por esse produto é:

$$\Delta(k(u-i)\times k(v-j)) = k(u-i)\times\Delta(k(v-j)) + k(v-j)\times\Delta(k(u-i)) \tag{4.15}$$

Como o valor resultante dos filtros de reconstrução é um valor normalizado do intervalo $[0,1]$ e tendo em conta a análise de incerteza da secção 4.4.3.1, os valores do produto dos filtros de reconstrução em ambas dimensões são apresentados na seguinte tabela:

Tabela 4.4: Incerteza do produto dos filtros de reconstrução

	$k_{nn}\times k_{nn}$	$k_{linear}\times k_{linear}$	$k_{spline}\times k_{spline}$
Float	2.38×10^{-7}	3.37×10^{-7}	1.38×10^{-3}
Double	4.44×10^{-16}	6.28×10^{-16}	5.96×10^{-8}

Como explicado em 4.4.3, a incerteza associada à função $I_s(\lfloor u\rfloor,\lfloor v\rfloor)$ é igual ao valor de $Erro(255)$ devido à restrição dos valores de intensidade de cor dos pixeis ao intervalo $[0,255]$. Assim, o cálculo do valor de intensidade de cor da interpolação por convolução de um dos pixeis da região da imagem original é dado por:

$$\begin{aligned}&\Delta(I_s(\lfloor u\rfloor,\lfloor v\rfloor)\times\Delta(k(u-i)\times k(v-j))) = \\ &I_s(\lfloor u\rfloor,\lfloor v\rfloor)\times\Delta(k(u-i)\times k(v-j)) + \Delta(I_s(\lfloor u\rfloor,\lfloor v\rfloor))\times k(u-i)\times k(v-j)\end{aligned} \tag{4.16}$$

Assim, o valor de incerteza do valor de intensidade de cor obtido pela convolução de um pixel é apresentado na seguinte ~~tabela:~~

Tabela 4.5: Incerteza do valor de intensidade de cor de um pixel da convolução

	$I_s\times k_{nn}\times k_{nn}$	$I_s\times k_{linear}\times k_{linear}$	$I_s\times k_{spline}\times k_{spline}$
Float	7.59×10^{-5}	1.01×10^{-4}	3.52×10^{-1}
Double	1.42×10^{-13}	1.89×10^{-13}	1.52×10^{-5}

Tendo em conta a operação de convolução sobre os a^2 pixeis da região de apoio de pixeis caraterística de cada filtro de reconstrução. O valor de incerteza do valor de intensidade de cor obtido na operação de convolução de um pixel da imagem reamostrado é apresentado na seguinte ~~tabela:~~

Tabela 4.6: Incerteza do valor de intensidade de cor de um pixel da convolução

	$\sum^a\sum^a I_s\times k_{nn}\times k_{nn}$	$\sum^a\sum^a I_s\times k_{linear}\times k_{linear}$	$\sum^a\sum^a I_s\times k_{spline}\times k_{spline}$
Float	1.51×10^{-4}	2.02×10^{-4}	1.408
Double	2.84×10^{-13}	3.78×10^{-13}	6.07×10^{-5}

4.4.3.3 Conclusão da Análise Numérica

A análise numérica do processo de reamostragem e redimensionamento permitiu traçar uma paralelização entre os dois tipos diferentes de dados utilizados para a representação de números de vírgula flutuante. Os tipos *float* e *double* têm um valor de incerteza associado aos resultados das operações onde são utilizados devido ao arredondamento necessário para a representação de um número real.

A tabela 4.6 expõe a incerteza associada aos resultados do processo de reamostragem e redimensionamento de imagem, atendendo ao tipo de dados utilizado e filtro de reconstrução utilizado.

Pelos resultados apresentados na tabela anterior é possível concluir que as diferenças de precisão da representação de números ~~de~~ vírgula flutuante não têm impacto no valor de intensidade de cor do pixel resultado da convolução, pois como os valores de intensidade de cor dos pixeis pertencem ao conjunto de números naturais incertezas inferiores a uma unidade não são suficientes para modificar os resultados.

Contudo, a utilização do tipo *float*, em especifico no processo de reamostragem e redimensionamento com o filtro de reconstrução por *spline*, apresenta um valor de incerteza superior a um. Essa ocorrência influencia os resultados do processo de reamostragem, provocando que o valor de intensidade de cor do pixel da imagem reamostrado calculado seja diferente do valor esperado. No entanto, devido às elevadas resoluções de imagem utilizadas atualmente a visão humana não tem percepção suficiente para distinguir as diferenças das caraterísticas e detalhes da imagem [42].