ACELERAÇÃO DE ALGORITMOS DE SUMARIZAÇÃO DE VÍDEOS COM PROCESSADORES GRÁFICOS (GPUS) E MULTICORE CPUS

SUELLEN SILVA DE ALMEIDA

ACELERAÇÃO DE ALGORITMOS DE SUMARIZAÇÃO DE VÍDEOS COM PROCESSADORES GRÁFICOS (GPUS) E MULTICORE CPUS

Dissertação apresentada ao Programa de Pós-Graduação em Ciência da Computação do Instituto de Ciências Exatas da Universidade Federal de Minas Gerais como requisito parcial para a obtenção do grau de Mestre em Ciência da Computação.

ORIENTADOR: ARNALDO DE ALBUQUERQUE ARAÚJO
COORIENTADOR: DAVID MENOTTI

Belo Horizonte
Agosto de 2014

Ficha catalografica elaborada pela Biblioteca do ICEx – UFMG

Almeida, Suellen Silva de

A448a Aceleração de Algoritmos de Sumarização de Vídeos com Processadores Gráficos (GPUs) e multicore CPUs / Suellen Silva de Almeida. — Belo Horizonte, 2014
xxii, 82 f. : il. ; 29cm

Dissertação (mestrado) — Universidade Federal de Minas Gerais - Departamento de Ciência da Computação

Orientador: Arnaldo de Albuquerque Araújo
Coorientador: David Menotti

1. Computação – Teses. 2. Processamento de imagens - Técnicas digitais – Teses. 3. Algoritmos paralelos – Teses. I. Orientador. II. Coorientador. III. Título.

CDU 519.6*84(043)

UNIVERSIDADE FEDERAL DE MINAS GERAIS
INSTITUTO DE CIÊNCIAS EXATAS
PROGRAMA DE PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO

FOLHA DE APROVAÇÃO

Aceleração de algoritmos de sumarização de vídeos com GPUs e multicore CPUs

SUELLEN SILVA DE ALMEIDA

Dissertação defendida e aprovada pela banca examinadora constituída pelos Senhores:

PROF. ARNALDO DE ALBUQUERQUE ARAÚJO - Orientador
Departamento de Ciência da Computação - UFMG

PROF. DAVID MENOTTI GOMES - Coorientador
Departamento de Computação - UFOP

PROF. HÉLIO PEDRINI
Instituto de Computação - UNICAMP

PROF. WILLIAM ROBSON SCHWARTZ
Departamento de Ciência da Computação - UFMG

Belo Horizonte, 22 de agosto de 2014.

A Deus, aos meus avôs, Guilherme e Pedro, aos meus pais, Nailto e Milena, a minha irmã, Priscilla, ao meu amor, Lúcio.

Agradecimentos

Sou muito grata a Deus, por nunca desistir de mim, por me fortalecer e se fazer presente em todos os momentos da minha vida; por me surpreender e me ensinar sempre. Agradeço aos meus pais, Nailto e Milena, pelo amor incondicional, dedicação, preocupação, apoio e incentivo. Serei eternamente grata por tudo que fizeram e fazem por mim. À minha irmã, Priscilla pela torcida e ajuda em momentos cruciais. Ao meu namorado, Lúcio, pelo amor, apoio e companheirismo e por entender o momento em que o mestrado foi a prioridade em minha vida. E às minhas amigas que sempre torceram por mim, em especial Bárbara, Natasha e Cecília, pela preocupação e incentivos constantes.

Agradeço ao Prof. Arnaldo de Albuquerque Araújo, pela orientação e incentivo e ao meu co-orientador, David Menotti, pelas ideias, sugestões e discussões essenciais para o desenvolvimento deste trabalho. Obrigada pela dedicação, cuidado e preocupação durante todo o mestrado.

Aos membros da banca, Hélio Pedrini e William Robson Schwartz pelas revisões e sugestões que contribuíram para este trabalho. Aos colegas Sandra Eliza Fontes de Avila e Edward Jorge Yuri Cayllahua Cahuina, por disponibilizarem as implementações de seus métodos sumarização de vídeos, utilizados como base deste trabalho. Aos colegas do Núcleo de Processamento Digital de Imagens (NPDI), pela vivência, conversas e sugestões. E aos colegas da UFMG, Antonio Carlos, Carlos Caetano e Victor Hugo.

Por fim, agradecimento especial às agências de fomento pelo suporte financeiro a esta dissertação: CNPq, Projeto InWeb, CAPES e FAPEMIG.

"Science without religion is lame,
religion without science is blind."
(Albert Einstein)

Resumo

A recente e rápida evolução das mídias digitais estimularam a criação, o armazenamento e a distribuição de dados, como por exemplo os vídeos, gerando um grande volume de dados e introduzindo a necessidade de um gerenciamento mais eficiente destes vídeos. Os métodos de sumarização de vídeos consistem em gerar resumos concisos do conteúdo desses vídeos e o resultado deste processo, ou seja, os resumos, permitem pesquisa, indexação e acesso mais rápido a grandes coleções de vídeos. No entanto, a maioria dos métodos de sumarização não apresenta um bom desempenho com vídeos de longa duração e de alta resolução. Uma forma de reduzir o grande tempo de execução é o desenvolvimento de algoritmos paralelos, aproveitando as vantagens das recentes arquiteturas de computadores que permitem alto paralelismo, como *Graphics Processor Units* (GPUs) e *multicore* CPUs. Este trabalho propõe a paralelização de dois métodos de sumarização de vídeos existentes na literatura. O primeiro é baseado na extração de características de cor dos quadros dos vídeos e no algoritmo de clusterização k-means, enquanto o segundo é baseado na segmentação temporal dos vídeos e em palavras visuais obtidas através de descritores locais. Para ambos os métodos, foram consideradas as seguintes implementações: GPUs, *multicore* CPUs e, finalmente, uma versão híbrida distribuindo as etapas dos métodos em GPUs e *multicore* CPUs a fim de maximizar o desempenho. Os experimentos foram realizados utilizando 240 vídeos variando a resolução dos quadros (320×240, 640×360, 1280×720 e 1920×1080 *pixels*) e a duração (1, 3, 5, 10, 20 e 30 minutos). Os resultados mostram que as implementações realizadas superam a versão sequencial dos dois métodos, mantendo a qualidade dos resumos.

Palavras-chave: Processamento de Vídeos, Sumarização de Vídeos, GPUs, multicore CPUs, Algoritmos Paralelos.

Abstract

The recent and fast evolution of digital media have stimulated the creation, storage and distribution of data, such as digital videos, generating a large volume of data and requiring efficient technologies to increase the usability of these data. Video summarization methods consist of generating concise summaries of video contents and it enable faster browsing, indexing and accessing of large video collections. However, these methods often perform slow with large duration and high quality video data. One way to reduce this long time of execution is to develop parallel algorithms, using the advantages of the recent computer architectures that allow high parallelism, i.e., Graphics Processor Units (GPUs) and multicore CPUs. This work proposes parallelizations of two video summarization methods. The former is based on color feature extraction from video frames and k-means clustering algorithm and the latter is based on temporal video segmentation and visual words obtained by local descriptors. For the two methods, some implementations were considered: GPUs, multicore CPUs, and ultimately a distribution of computations steps onto both hardware to maximise performance. The experiments were performed using 240 videos varying frame resolution (320×240, 640×360, 1280×720 e 1920×1080 pixels) and video length (1,3,5,10,20 and 30 minutes). The results shows that the implementations overcome the sequential version of both methods, keeping the quality of the summaries.

Keywords: Video Processing, Video Summarization, GPUs, multicore CPUs, Parallel Algorithms.

Lista de Figuras

Lista de Tabelas

Sumário

Capítulo 1

Introdução

Grandes coleções de dados multimídia, como áudios, imagens e vídeos, são criadas diariamente em consequência da evolução e popularização de dispositivos para aquisição e armazenamento de dados, *i.e., smartphones* e câmeras fotográficas. Outro fator que contribui com a geração desses dados é a disponibilidade de meios de transmissão de dados em alta velocidade, como sites de compartilhamento de vídeos e canais de televisão em alta definição.

A criação de grandes coleções de dados gera a necessidade de gerenciamento e armazenamento eficientes e eficazes de todas essas informações, juntamente com um fácil acesso aos dados armazenados. Os vídeos possuem grandes quantidades de quadros e é necessário uma taxa de 25 quadros por segundo para que os seres humanos não percebam descontinuidades. Sendo assim, um vídeo de 5 minutos gera 7.500 imagens. Esse grande volume de dados é um obstáculo para várias aplicações, levando a uma forte demanda por técnicas que permitam ao usuário ter noção do vídeo sem assistir ao mesmo por inteiro [Truong & Venkatesh, 2007]. Com esta finalidade, novos métodos e técnicas têm sido estudados e a sumarização automática de vídeos é um assunto amplamente pesquisado [Gianluigi & Raimondo, 2006; Mundur et al., 2006; Truong & Venkatesh, 2007; Gao et al., 2009; Avila et al., 2011; Almeida et al., 2012; Guan et al., 2012; Li et al., 2013; Evangelio et al., 2013; Cahuina et al., 2013; Almeida et al., 2014], devido a sua importância na indexação, navegação e recuperação de vídeos, assim como no pré-processamento em diversas aplicações.

Sumarização de vídeos é o processo para geração de um pequeno resumo de um vídeo, preservando a mensagem original deste vídeo [Truong & Venkatesh, 2007]. Ou seja, dado um vídeo digital, deve-se selecionar, um conjunto conciso de quadros que seja representativo. Esse resumo pode ser uma coleção de imagens, organizada de forma sequencial, conhecido como *representative frames* ou *static story-*

board, ou mesmo um vídeo com as principais partes do vídeo original, conhecido como *dynamic summary* ou *moving storyboard*.

O tempo de execução de algumas abordagens, como Cahuina et al. [2013], pode atingir, em uma arquitetura particular, o dobro do tempo de execução do vídeo para criação do resumo, o que é indesejado. Um dos objetivos da sumarização de vídeos é a possibilidade de manipular uma grande quantidade de dados com maior facilidade. Se o algoritmo de sumarização gasta mais tempo que a duração do vídeo para gerar o resumo, essa característica não é obtida. Dessa forma, é desejável que, além de gerar resumos concisos, o algoritmo de sumarização de vídeos seja rápido.

Ainda neste contexto, com os avanços nas representações dos vídeos e imagens digitais, também surgiu a necessidade de *hardware* e *software* que realizem com eficiência o processamento e manipulação destes dados. Duas arquiteturas de computadores podem ser citadas para tal processamento. A primeira consiste nas unidades centrais de processamento com múltiplos núcleos, cada vez mais presentes em computadores e *notebooks*. Já a segunda são as unidades de processamento gráfico (GPUs), compostas por centenas de elementos de processamento e presentes nas placas de vídeos recentes.

Por um lado, os *chips* dos processadores gráficos iniciaram como processadores de vídeo com função fixa, porém, com o tempo, tornaram-se altamente programáveis e poderosos em termos de computação, sendo utilizados para fins gerais de computação. Em 2006, a empresa NVIDIA introduziu $CUDA^{TM}$, uma arquitetura que aproveita a computação paralela das GPUs NVIDIA para resolver muitos problemas computacionais complexos de forma mais eficiente do que em CPU. A arquitetura CUDA vem com um ambiente de *software* que permite aos desenvolvedores o uso da linguagem C como uma linguagem de programação de alto nível [NVIDIA, 2014b]. Por outro lado, as CPUs com múltiplos núcleos são mais eficientes que as GPUs na execução de várias tarefas com conteúdos diversos. Além disso, podem lidar com programas complexos com mais facilidade, porém, ainda ficam atrás das GPUs quando grandes quantidades de cálculos simples podem ser realizadas em paralelo [Palacios & Triska, 2011].

Finalmente, o problema de gerar resumos de vídeos de forma rápida, independentemente do tamanho e resolução dos vídeos, pode ser resolvido através do alto poder de processamento de CPUs *multicore* e GPUs. Este trabalho apresenta propostas para processamento paralelo adequadas à implementação em placas gráficas e CPUs com múltiplos núcleos para os dois métodos de sumarização de vídeos citados anteriormente [Avila et al., 2011] e Cahuina et al. [2013]. Para avaliar a eficiência das implementações propostas, foi criada um conjunto com 240 vídeos de diversos

tamanhos, resoluções e diferentes gêneros.

Essas duas abordagens de sumarização de vídeos foram escolhidas por serem recentes e gerarem resumos de alta qualidade. Especificamente, o método proposto em Avila et al. [2011] (VSUMM) apresentou resultados com qualidade superior a abordagens de sumarização de vídeos consideradas estado da arte [Mundur et al., 2006; Furini et al., 2007], além de ser citado e comparado diversas vezes com novos métodos relacionados à sumarização de vídeos. Já o método proposto em Cahuina et al. [2013] apresentou resultados ainda melhores que os da abordagem VSUMM, atingindo 10% de melhoria em relação à acurácia do método, e consequentemente, também apresentou resumos com qualidade superior aos métodos considerados estado da arte. Além disso, ambos métodos são de autores conhecidos e que disponibilizaram o código para utilização neste trabalho.

1.1 Motivação

A evolução e popularização dos dispositivos de criação e distribuição de vídeos, assim como a evolução da computação paralela, motivam este trabalho no desenvolvimento de algoritmos paralelos ágeis de sumarização de vídeos para gerenciamento de grandes quantidades de vídeos. A necessidade de técnicas para o acesso rápido às informações contidas nos vídeos é evidente devido ao grande aumento do volume de dados, bem como obter de modo ágil uma noção dos principais objetos e eventos presentes nos vídeos. Resumos gerados de forma concisa, inteligente e paralela facilitarão o acesso do usuário a um volume substancial de vídeos de maneira eficiente e eficaz, aumentando sua produtividade.

1.2 Objetivos

1.2.1 Objetivo Geral

Paralelizar de forma eficiente dois algoritmos de sumarização automática de vídeos [Avila et al., 2011; Cahuina et al., 2013] que geram resumos estáticos concisos e representativos de forma que a elaboração dos resumos gaste o menor tempo possível. Esta paralelização será realizada através de CPUs com múltiplos núcleos e unidades de processamento gráfico (GPUs), utilizando a arquitetura CUDA (*Compute Unified Device Architecture*) e linguagem CUDA C (linguagem C para CUDA).

1.2.2 Objetivos Específicos

1. Propor algoritmos paralelos e eficientes para duas abordagens de sumarização automática de vídeos propostas em Avila et al. [2011] e Cahuina et al. [2013].

2. Criar um conjunto com vídeos de diversas durações, *i. e.*, 1, 3, 5, 10, 20 e 30 minutos e resoluções, *i. e.*, 320×240, 640×360, 1280×720 e 1920×1080 *pixels*.

3. Comparar as versões paralelas de ambas as abordagens, destacando facilidades e dificuldades de paralelizar determinados tipos de algoritmos.

4. Analisar a escolha mais apropriada entre GPU e *multicore* CPU em relação a diversos algoritmos.

1.3 Contribuições

A principal contribuição deste trabalho é a paralelização de métodos para a elaboração automática de resumos estáticos de vídeos. A paralelização resultará no aceleração da criação de resumo para vídeos com qualquer duração e resolução. A metodologia proposta integra as mais modernas técnicas e funções presentes na arquitetura CUDA para GPU e as bibliotecas mais eficientes para utilização de múltiplos núcleos da CPU. Além disso, na literatura, não foram encontrados algoritmos paralelos para sumarização automática de vídeos.

Outra contribuição importante é a disponibilização da ferramenta de sumarização paralela para a comunidade acadêmica. Algoritmos de processamento de imagens e vídeos e de visão computacional geralmente são computacionalmente intensivos e repetitivos, o que pode torná-los lentos. Com isso, o paralelismo é uma das formas mais adequadas de acelerá-los, principalmente utilizando as modernas GPUs com alto poder computacional.

1.4 Organização do Documento

Esta dissertação está organizada da seguinte forma. O Capítulo 2 fornece os fundamentos básicos para a compreensão dos métodos e algoritmos utilizados e mencionados neste trabalho. O Capítulo 3 apresenta uma revisão bibliográfica sobre os principais métodos de sumarização de vídeos e métodos de processamento de imagens e vídeos implementados de forma paralela. O Capítulo 4 descreve a metodologia desenvolvida neste trabalho para implementação dos algoritmos paralelos

para sumarização de vídeos. No Capítulo 5, são apresentados o conjunto de vídeos utilizado nos experimentos e os resultados obtidos. Finalmente, o Capítulo 6 apresenta as conclusões deste trabalho.

Capítulo 2

Fundamentação Teórica

Este capítulo apresenta os fundamentos teóricos utilizados na elaboração deste trabalho. Na Seção 2.1, são apresentados os conceitos relacionados à sumarização automática de vídeos, além de duas abordagens para criação de resumos estáticos [Avila et al., 2011; Cahuina et al., 2013], que foram utilizadas como base para o este trabalho. Os fundamentos utilizados para paralelização dessas abordagens são descritos na Seção 2.2. Essa seção explica alguns conceitos gerais de computação paralelas, bem como detalhes de bibliotecas utilizadas em algoritmos *multicore* CPU, e a arquitetura/linguagem CUDA, utilizada para paralelizar algoritmos em GPU.

2.1 Sumarização Automática de Vídeos

A sumarização automática de vídeos é um processo de criação de uma pequena versão, um resumo, do conteúdo original de um vídeo, cujo objetivo é fornecer ao usuário uma informação concisa e útil do seu conteúdo, preservando a mensagem original do vídeo. Segundo Truong & Venkatesh [2007], existem dois formatos principais de resumo de vídeos: *keyframes* ou *video skin*. O primeiro formato, também conhecido como *representative frames* e *static storyboard*, consiste em uma coleção de imagens salientes, os *keyframes*, extraída do vídeo original, resultando em resumos estáticos. Já o segundo formato, também conhecido como *moving storyboard* e *summary sequence*, consiste em uma coleção de segmentos/tomadas de um vídeo (e o áudio correspondente), também extraída do vídeo original, resultando em resumos dinâmicos. Os resumos dinâmicos possuem áudio e elementos em movimento, características que deixam o resumo com mais expressividade e mais interessante de assistir do que os resumos estáticos. Por outro lado, os resumos estáticos apresentam maiores possibilidades de organização para busca e navegação, em vista da

apresentação estritamente sequencial dos resumos dinâmicos. Ambos algoritmos de sumarização automática de vídeos utilizados como base para este trabalho [Avila et al., 2011; Cahuina et al., 2013] possuem foco nos resumos estáticos.

De maneira genérica, a tarefa de sumarização de vídeos para geração de resumos estáticos se inicia com a segmentação do vídeo, seguida pela extração dos quadros-chave, e por fim a detecção de redundâncias entre os quadros-chave detectados. A segmentação temporal do vídeo é geralmente o primeiro passo para a sumarização de vídeos e tem como objetivo separar o vídeo em seus componentes básicos como tomadas e quadros. A segmentação em tomadas é largamente utilizada em diversos trabalhos como Gianluigi & Raimondo [2006]; Cámara-Chávez et al. [2007]; Cahuina et al. [2013]. Neste tipo de segmentação, é necessário detectar os efeitos visuais dos vídeos como os cortes que representam a transição entre duas tomadas e outros efeitos chamados de transições graduais (*fade-out*, *fade-in*, *dissolve*, *wipe*). Outro tipo de segmentação é a separação do vídeo em quadros, aplicada em Mundur et al. [2006]; Furini et al. [2007]; Avila et al. [2011]. Neste caso, cada quadro é tratado separadamente e não ocorre a análise temporal do vídeo. Pode-se analisar todos os quadros do vídeo ou utilizar uma amostragem dos quadros.

Para extrair os quadros-chave, muitos algoritmos de sumarização de vídeos utilizam descritores locais ou globais para descrever o conteúdo visual de cada quadro do vídeo. Os descritores podem ser baseados em informações de cor, forma, textura, ou podem ser locais como SIFT (*Scale-Invariant Feature Transform*) [Lowe, 1999], SURF (*Speeded-Up Robust Features*) [Bay et al., 2008] e HoG (*Histogram of Oriented Gradients*) [Dalal & Triggs, 2005], entre outros. Estes descritores extraem as características dos quadros, gerando vetores de características. Um vetor de características caracteriza de forma única um determinado objeto, facilitando a distinção entre imagens visualmente diferentes, de acordo com alguma métrica. A partir desses vetores, os quadros podem ser agrupados de acordo com a similaridade reportada pelas características extraídas. O ponto mais próximo do centróide de cada grupo é considerado o ponto mais representativo do grupo, gerando assim o conjunto de quadros-chave do vídeo. O processo de agrupamento pode ser classificado como hierárquico (separações de grupos baseado na similaridade entre os padrões) ou particional (identificação das partições por meio de otimização de uma função de custo). Dentre os diversos algoritmos de agrupamento estão o *k-means* [Macqueen, 1967] e o *x-means* [Pelleg & Moore, 2000].

Após a extração dos quadros-chave, podem existir quadros muito parecidos e redundantes, gerando a necessidade de uma filtragem dos quadros representativos. Para isso, alguma medida de similaridade é aplicada entre pares de características

extraídas para obter o grau de similaridade entre dois quadros. Se forem muito semelhantes, um destes quadros é removido dos quadros representativos. Dentre as medidas de similaridade estão a distância Euclidiana, distância de Manhattan e similaridade de cossenos.

2.1.1 Sumarização Automática de Vídeos Proposta em Avila et al. [2011]

Avila et al. [2011] propõem uma abordagem simples e efetiva para sumarização automática de vídeos, chamada VSUMM, baseada na extração de características de cor e na classificação não-supervisionada. A Figura 2.1 apresenta as principais etapas do método.

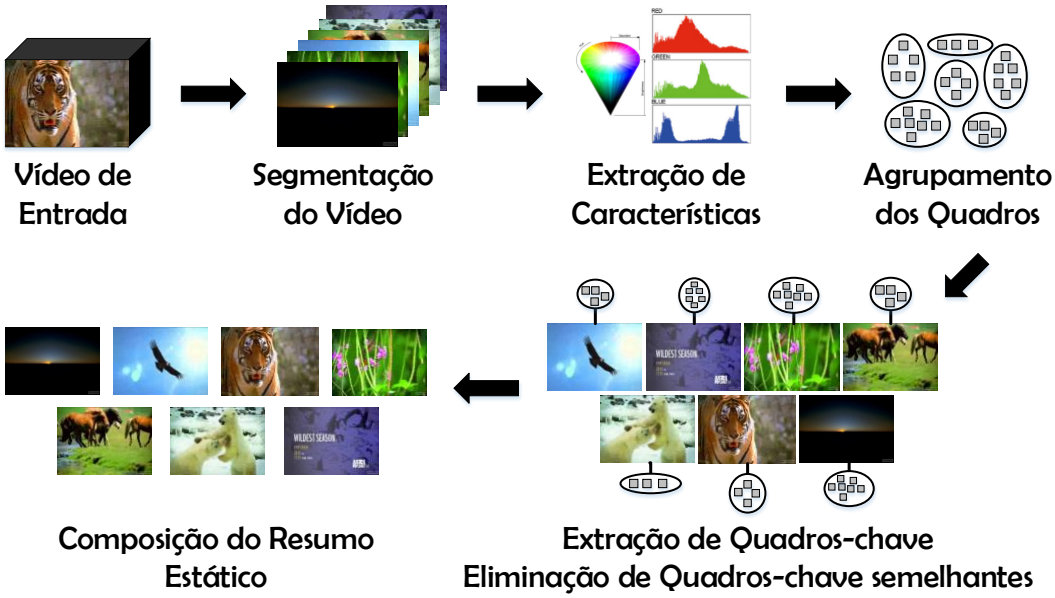


Figura 2.1: Diagrama da Abordagem de Sumarização de Vídeos proposta em Avila et al. [2011]

Segmentação do Vídeo

A primeira etapa desta abordagem de sumarização automática de vídeos é a segmentação temporal do vídeo e existem diversos tipos diferentes de métodos que realizam esta função. Nesta abordagem, a segmentação adotada consiste em separar o vídeo em quadros. A taxa de quadros por segundo necessária para que os seres humanos não percebam descontinuidades nos vídeos é de pelo menos 25 quadros

por segundo. Estes quadros, mostrados a cada segundo contêm informações redundantes e por isso não há necessidade de analisar todos os quadros do vídeo. Dessa forma foi escolhida a amostragem para segmentar um quadro por segundo.

Extração de Características e Eliminação de Quadros Inexpressivos

Nesta etapa, o histograma de cor é utilizado para descrever o conteúdo visual dos quadros do vídeo, representando a distribuição da frequência de ocorrência dos valores cromáticos em uma imagem. Os quadros são transformados do espaço de cores RGB para o HSV, o qual é uma escolha comum para manipulação de cores [Avila et al., 2011]. O histograma de cor é computado apenas para o componente *Hue*, que representa o componente de cor espectral dominante em sua forma pura [Avila et al., 2011]. Por fim, a quantização do histograma de cor foi definida com 16 *bins*, com o objetivo de reduzir significantemente a quantidade de dados a serem processados, sem perda de informações importantes [Avila et al., 2011].

Antes da computação do histograma de cor do componente *hue*, os quadros monocromáticos originados de efeitos de *fade-in/out* são eliminados pois são considerados inexpressivos. De fato, este processo é feito pela análise do desvio padrão do histograma RGB, como descrito em Avila [2008].

Agrupamento dos Quadros

O algoritmo de clusterização escolhido é o *k-means* [Macqueen, 1967], um dos algoritmos de aprendizado não-supervisionado mais simples e amplamente utilizado, apresentando resultados promissores para o problema em questão. A principal ideia do *k-means* é simultaneamente maximizar a distância inter-grupos e minimizar a distância intra-grupo.

Inicialmente, o algoritmo distribui os quadros do vídeo em k grupos, em que o valor de k é determinado *a priori*. O próximo estágio é computar a média dos elementos em cada grupo, que corresponde aos centróides iniciais. Então o grupo de cada elemento é encontrado através da proximidade com os centróides. Com esta nova organização dos elementos, os novos centróides são calculados e este processo é repetido até a convergência. A convergência do *k-means* é normalmente obtida quando os grupos se estabilizam.

Extração de Quadros-chave

Dado que os grupos de quadros foram formados pelo algoritmo *k-means*, o próximo passo é identificar os quadros-chave. Os grupos nos quais a quantidade de quadros

é menor do que a metade da média do tamanho dos grupos não são considerados para escolha dos quadros-chave Nos grupos restantes, chamados de grupos-chave, o quadro mais próximo do centróide é selecionado como o quadro mais representativo de cada grupo. Para medir a proximidade entre os quadros e o centróide é utilizada a distância Euclidiana.

Eliminação de Quadros-chave semelhantes

Por fim, é realizada uma filtragem nos quadros-chave com o objetivo de eliminar os semelhantes. Para isso, todos os quadros-chave são comparados entre si de acordo com o histograma de cor. Se a distância Euclidiana entre dois quadros-chave é menor que um limiar τ, então o quadro-chave é eliminado do resumo. Finalmente, os quadros-chave resultantes são dispostos em ordem cronológica para facilitar o entendimento visual do resumo.

2.1.2 Sumarização Automática de Vídeos Proposta em Cahuina et al. [2013]

Cahuina et al. [2013] propõem uma abordagem eficaz para sumarização automática de vídeos, baseada na segmentação temporal do vídeo e em palavras visuais obtidas a partir de descritores locais. A Figura 2.2 apresenta as principais etapas deste método.

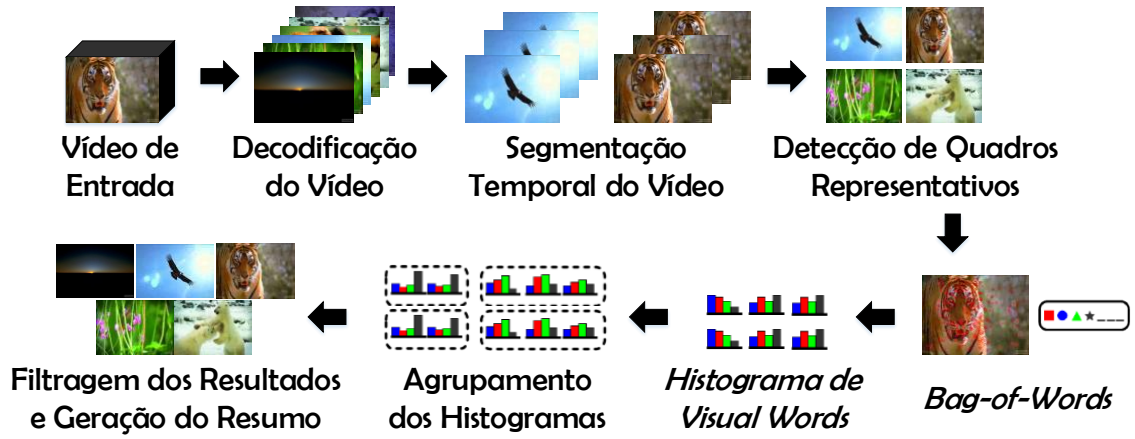


Figura 2.2: Diagrama da Abordagem de Sumarização de Vídeos proposta em Cahuina et al. [2013]

Decodificação do Vídeo

Esta etapa consiste em decodificar o vídeo de entrada em quadros que serão utilizados para posterior processamento. Esse passo é similar à segmentação do vídeo da abordagem anterior, porém todos os quadros do vídeos são extraídos, não apenas um por segundo como na outra abordagem.

Segmentação Temporal do Vídeo

O objetivo desta etapa é extrair os segmentos dos vídeos com informações mais relevantes para posterior análise. Para isso, inicialmente, é analisada a medida de similaridade de cosseno computada entre os histogramas de cor RGB de quadros consecutivos. Valores de dissimilaridade altos (limiar 0.4 [Cahuina et al., 2013]) implicam que transições de vídeo de corte (mudança instantânea de uma tomada para outra) foram detectadas.

Ao analisar o vetor de dissimilaridade de cosseno, é comum encontrar transições de corte falsas [Cahuina et al., 2013]. Para solucionar este problema, um procedimento de filtragem é aplicado aos vetores de dissimilaridade do vídeo. Este procedimento consiste em escolher cada valor do vetor de dissimilaridade como pivô e analisar os vizinhos. Então é encontrado o maior valor entre os vizinhos (mv) e se o valor do pivô (v) é maior do que mv, o valor do pivô é modificado de acordo com a seguinte fórmula:

$$s_i = \frac{v_i - mv_i}{v_i} \tag{2.1}$$

onde i é a posição no vetor de dissimilaridade, s_i é o valor computador, v_i é o valor do pivô e mv_i é o valor máximo da vizinhança do pivô. Dessa forma o vetor de dissimilaridade é refinado e os picos são reconhecidos como mudanças abruptas. O número de segmentos presente no vídeo é a quantidade de mudanças abruptas que foram detectadas.

Os segmentos detectados a partir do vetor de dissimilaridade podem conter efeitos visuais como transições de *fade-in/out* e dissolução. Essas transições são indesejáveis para o processamento do vídeo, pois as imagens destes efeitos podem ficar distorcidas e inexpressivas. Por isso, as partes do vídeo onde esses efeitos ocorrem são detectadas e eliminadas de processamentos posteriores. Essa eliminação ocorre através de uma análise das variâncias de todos os quadros do vídeo. Os efeitos indesejáveis estão, normalmente, localizados em área de vale do vetor de variâncias, *i. e.*, pontos de mínimo local, e o procedimento descrito em Won et al. [2003] e em

Cámara-Chávez et al. [2007], é o utilizado para encontrar estes vales. Finalmente são obtidas as tomadas do vídeo sem efeitos de transição.

Detecção de Quadros Representativos

Depois da segmentação do vídeo, o método aplica o algoritmo de clusterização *x-means* [Pelleg & Moore, 2000] para detectar os quadros mais representativos para cada parte do vídeo segmentado, utilizando os histogramas de cor. Os quadros mais próximos aos centróides dos grupos criados pelo algoritmo de agrupamento são considerados como os quadros representativos de cada segmento do vídeo. O algoritmo *x-means* é uma extensão do *k-means* [Macqueen, 1967], onde a quantidade de grupos k é rapidamente estimada. O usuário indica os limites mínimo e máximo para o valor de k e o algoritmo utiliza este intervalo para selecionar o melhor conjunto de centróides e o melhor valor de k.

Bag of Visual Words

Para representar o conteúdo semântico dos vídeos, a abordagem de *Bag-of-Words* (BoW) é adotada [Yang et al., 2007; Van Gemert et al., 2010], considerando uma imagem como um documento. As "palavras" são as entidades visuais presentes na imagem, ou seja, características da imagem. Utilizando esta informação uma sumarização semântica baseada em objetos do vídeo é realizada. A abordagem de *Bag-of-Visual-Words* consiste em três operações: detecção de características, descrição de características e geração do vocabulário de "palavras visuais". Para as duas primeiras operações, o método de sumarização pode utilizar qualquer descritor local como SIFT [Lowe, 1999], SURF [Bay et al., 2008] e HoG [Dalal & Triggs, 2005].

O vocabulário de "palavras visuais", também conhecido como "codebook" é gerado a partir dos vetores de características obtidos durante o processo de detecção e descrição de características. Ou seja, cada palavra visual, ou "codeword", representa um grupo de várias características similares. Para computar as "codewords" todos os vetores de características são agrupados e classificados utilizando um algoritmo de clusterização, como o *Linde-Buzo-Gray* (LBG) [Linde et al., 1980]. Dessa forma, as "codewords" são as palavras mais próximas aos centróides de cada grupo gerado na clusterização.

Histograma de *Visual Words*

Essa etapa é responsável por criar a distribuição de frequências das palavras visuais do vocabulário, ou seja, do histograma de palavras visuais. Isto é feito utilizando as

características locais das imagens para verificação de quais palavras visuais ocorrem em cada imagem (quadro representativo). As ocorrências são contadas e armazenadas em um vetor para cada quadro. Como resultado, cada quadro representativo terá um vetor de ocorrências de palavras visuais associado.

Clusterização dos Vetores de *Visual Words*

Neste estágio, o método utiliza os vetores de ocorrências de palavras visuais computado anteriormente e aplica o algoritmo *x-means*. Consequentemente, quadros como entidades visuais similares são transferidos para o mesmo grupo. Então, o quadro mais próximo ao centróide de cada grupo é escolhido como quadro-chave. Finalmente, os quadros-chave detectados são ordenados de acordo com a ordem temporal e o resumo estático é gerado.

Filtragem dos Resultados

Um passo adicional é realizado para eliminação de quadros-chave similares. Para isso, é computada a distância de Manhattan do histograma de cor de quadros consecutivos. Se for detectada alta similaridade entre dois histogramas, o quadro-chave é eliminado.

2.2 Computação Paralela

Processadores baseados em uma única unidade de processamento (CPU) aumentaram o desempenho e reduziram custo em aplicações de computadores por mais de duas décadas. No entanto, este processo chegou ao limite por volta de 2003, devido aos problemas de dissipação de calor e consumo de energia. Esses problemas limitam o aumento de frequências de *clock* de CPU e o número de tarefas que podem ser realizadas dentro de cada período de *clock*. A solução adotada pela indústria de processadores foi mudar para um modelo de processadores com várias unidades de processamento, conhecidas como núcleos [Diaz et al., 2012]. Essa evolução das arquiteturas de computadores com um grande número de núcleos (*cores*) aumentou a importância e as pesquisas na área da computação paralela, confirmando que o paralelismo é uma forma muito eficiente de acelerar diversos algoritmos [Navarro et al., 2013].

Com a criação de processadores com vários núcleos, duas abordagens principais podem ser citadas: *multi-core* e *many-core*. A primeira abordagem, *multi-core*,

integra alguns poucos núcleos em um único processador, permitindo a execução simultânea de programas sequenciais com a mesma velocidade. A maioria dos computadores de mesa e *notebooks* em uso atualmente incorporam esse tipo de processador. A segunda abordagem, *many-core*, utiliza uma grande quantidade de núcleos e destaca-se pela grande quantidade de dados que podem ser processados simultaneamente. Essa abordagem é exemplificada através das unidades de processamento gráfico (GPUs) [Diaz et al., 2012].

Devido a essa evolução dos processadores, a computação paralela ficou em evidência pois programas/algoritmos sequenciais não utilizam as vantagens e benefícios destes processadores. O desafio principal é desenvolver aplicações paralelas que escalam os problemas da melhor maneira possível entre o crescente número de processadores, da forma mais transparente possível. A ideia principal da computação paralela é que problemas grandes e complexos podem ser divididos em problemas menores e estes serem resolvidos de forma paralela utilizando vários processadores.

2.2.1 Modelos de Arquitetura Paralela

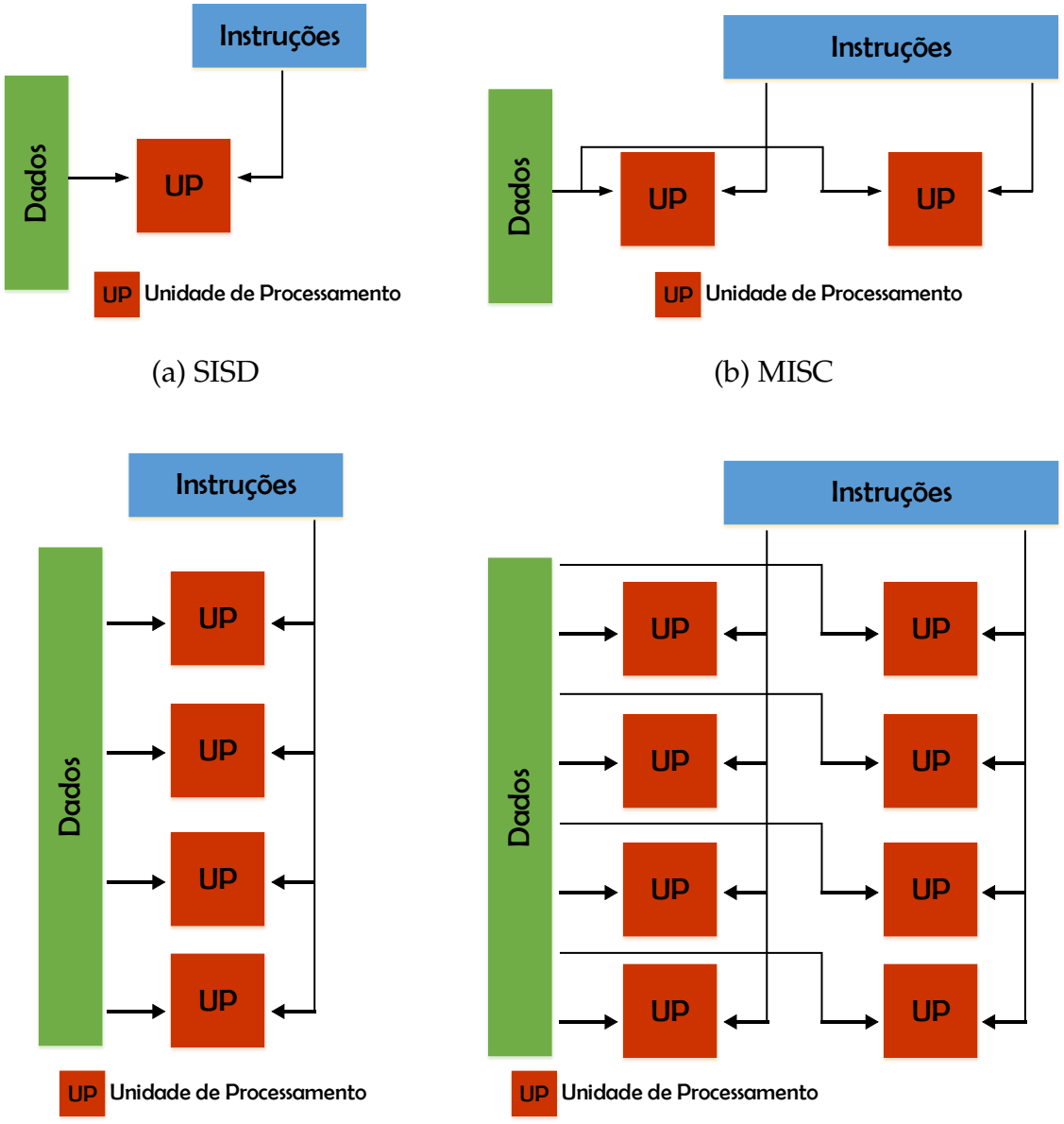
Segundo Flynn [1972], arquiteturas de computadores paralelos podem ser classificadas em quatro categorias, *i.e.*, SISD (*Single Instruction Single Data*), SIMD (*Single Instruction Multiple Data*), MISD (*Multiple Instruction Single Data*) e MIMD (*Multiple Instruction Multiple Data*). Esses modelos serão, resumidamente, apresentados a seguir e ilustrados na Figura 2.3

SISD (*Single Instruction Single Data*) : Esse modelo não explora o paralelismo, cada instrução opera sobre um único fluxo de dados e corresponde a computadores sequenciais convencionais.

SIMD (*Single Instruction Multiple Data*) : Descreve um método de operação de computadores com várias unidades de processamento em computação paralela. Nesse modelo, a mesma instrução é aplicada simultaneamente a diversos dados para produzir mais resultados. Arquiteturas de GPUs se enquadram nessa classificação.

MISD (*Multiple Instruction Single Data*) : Nessa arquitetura, múltiplos fluxos de instrução executam operações diferentes sobre os mesmos dados. Não há muitos exemplos da existência desse modelo de arquitetura, ao contrário do SIMD e do MIMD.

MIMD (*Multiple Instruction Multiple Data*) : Essa classificação diz respeito a unidades de processamento que executam múltiplas instruções diferentes sobre outros múltiplos conjuntos de dados. A maioria das arquiteturas paralelas está classificada na categoria MIMD, como os supercomputadores atuais e computadores pessoais com processadores *multicores*.



(a) SISD (b) MISC

(c) SIMD (d) MIMD

Figura 2.3: Modelos de Arquitetura SISD, MISD, SIMD e MIMD.

2.3 Computação em CPU *multicore*

Durante muito tempo, um dos métodos importantes para melhoria do desempenho dos dispositivos de computação foi aumentar a frequência do *clock* dos processadores e a quantidade de tarefas que podem ser executadas dentro de um período de *clock*. Nos últimos anos, no entanto, os fabricantes foram forçados a procurar alternativas dessa tradicional fonte de aumento do poder computacional, devido às restrições de energia e aquecimento e ao tamanho dos transistores, chegando ao seu limite físico. Os fabricantes, então, começaram a produzir processadores com vários núcleos ao invés de apenas um, uma grande evolução na indústria de processadores [Diaz et al., 2012]. A computação em CPU *multicore* consiste em utilizar esses núcleos dos processadores paralelamente para computação de uma ou várias tarefas simultaneamente. O modelo de arquitetura das CPUs *multicore* é o MIMD (*Multiple Instruction Multiple Data*), descrito na Subseção 2.2.1, onde os vários processadores podem operar de forma cooperativa ou concorrente, na execução de um ou vários aplicativos.

Dois artifícios muito utilizados para implementação na linguagem C/C++ de algoritmos utilizando os vários processadores da CPU são a API OpenMP e a classe Threads, explicados com mais detalhes a seguir.

2.3.1 OpenMP

OpenMP é uma interface de programação de aplicação (API) para algoritmos paralelos utilizando memória compartilhada em programas C, C++ e Fortran. Essa API consiste em um conjunto de diretivas de compilação, biblioteca de funções e variáveis de ambiente que permitem aos usuários a criação de programas paralelos através da implementação automática e otimizada de um conjunto de *threads* [Kasim et al., 2008; OpenMP, 2013]. O padrão OpenMP não é uma linguagem de programação, representa um padrão que define como os compiladores devem gerar códigos paralelos através da incorporação, nos programas sequenciais, de diretivas que indicam como o trabalho deve ser dividido entre os processadores.

Modelo de Programação

No OpenMP, a paralelização é explicitamente realizada com múltiplas *threads* dentro de um mesmo processo. A criação de *threads* é uma forma de um processo que divide a si mesmo em duas ou mais tarefas, que podem ser executadas simultaneamente. Cada *thread* possui sua própria pilha de execução, porém compartilha o

mesmo endereço de memória com as outras *threads* do mesmo processo (enquanto que cada processo possui seu próprio espaço de memória).

O modelo de programação inicia com uma única *thread* que executa sozinha as instruções até encontrar uma região paralela (identificada pela diretiva). Ao encontrar essa região, ela cria um grupo de *threads* que, juntas, executam o código dentro dessa região. Quando as *threads* completam a execução do código na região paralela, elas sincronizam-se e somente a *thread* inicial segue na execução do código até que uma nova região paralela seja encontrada ou que o programador decida encerrar essa *thread* [Sena & Costa, 2008].

2.3.2 Threads C++

A linguagem de programação C++, mais especificamente em sua versão *C++11 Standard*, oferece suporte a programas *multithread* através da classe *thread*. Duas formas de paralelismo podem ser implementadas com essa biblioteca. A primeira forma consiste em dividir uma aplicação em múltiplos e separados processos *single-threaded*, de forma que todos são executados ao mesmo tempo. A segunda forma executa múltiplas *threads* em um único processo. Mesmo que as *threads* executem independentemente das outras, e cada uma possa executar uma sequência de instruções diferentes, todas as *threads* de um processo compartilham o mesmo endereço de espaço, e a maioria dos dados pode ser acessado diretamente por todas elas. Essa biblioteca inclui classes para manipulação de *threads*, para proteger dados compartilhados, para sincronização de operações entre *threads* e para operações atômicas [Williams, 2012].

2.4 Computação em GPU

Computação GPU é a utilização da unidade de processamento gráfico (GPU) para resolução de problemas de computação, sem a restrição de problemas do contexto gráfico. A computação GPU oferece desempenho de alto nível para aplicativos ao transferir as partes de processamento intensivo do aplicativo para a GPU, enquanto o resto do código continua sendo executado pela CPU. Da perspectiva do usuário, os aplicativos simplesmente são executados com uma velocidade extra significativa.

CPU e GPU são uma combinação poderosa porque as CPUs consistem em alguns núcleos otimizados para processamento serial, enquanto as GPUs consistem em milhares de núcleos menores e mais eficientes, projetados para desempenho paralelo. Partes seriais do código são executadas pela CPU, enquanto as partes parale-

las são executadas pela GPU. A Figura 2.4 apresenta uma comparação arquitetural entre CPU e GPU. É possível ver que na GPU mais transistores são destinados ao processamento de dados ao invés de cache de dados e controle de fluxo, o que faz o processamento ser muito mais rápido do que em CPU.

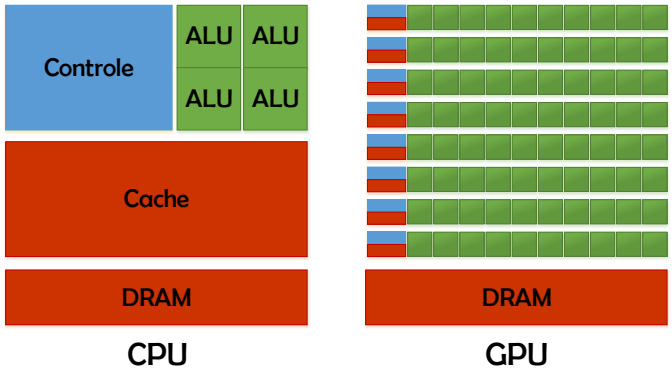


Figura 2.4: Comparação entre CPU e GPU [NVIDIA, 2014b].

Mais especificamente, a GPU é especialmente adequada para resolver os problemas que podem ser expressos como computações de dados paralelos, ou seja, o mesmo programa pode ser executado em muitos elementos em paralelo. Como o mesmo programa é executado para cada elemento dos dados, existe uma menor necessidade de controle de fluxo sofisticado, e, como é executado em vários elementos dos dados e possui uma intensidade aritmética elevada, a latência de acesso à memória pode ser escondida com cálculos, em vez de grandes caches de dados [NVIDIA, 2014b].

2.4.1 CUDA

Em 2006, a empresa NVIDIA introduziu $CUDA^{TM}$, uma arquitetura de computação paralela que aproveita a computação paralela das GPUs NVIDIA para resolver muitos problemas computacionais complexos de forma mais eficiente do que em CPU. O modelo de programação paralela CUDA foi criado para solucionar o problema de escalar programas entre vários núcleos da forma mais transparente possível, ao mesmo tempo que facilita a programação para desenvolvedores através da linguagem C.

Cada processador CUDA suporta o modelo SIMD (*Single Instruction Multiple Data*), descrito da Subseção 2.2.1, onde todas as *threads* concorrentes são baseadas

no mesmo código. No entanto, elas precisam seguir o mesmo caminho de execução [Ryoo et al., 2008].

Modelo de Programação

A linguagem utilizada no modelo de programação CUDA é a CUDA C, uma extensão da linguagem C com funções para manipulação de dados em GPU. Nesse modelo, existem funções chamadas *kernels* que são executadas N vezes em paralelo por N diferentes *threads* CUDA. Essas *threads* podem ter uma, duas ou três dimensões e formam um bloco de *threads* que também pode ter uma, duas ou três dimensões. Nas GPUs mais recentes, cada bloco pode ser formado por até 1024 *threads*. Esta configuração permite a computação em GPU de elementos como vetor, matriz ou volume. Os blocos são organizados em um *grid* que também pode ter até três dimensões. A quantidade de blocos do *grid* é normalmente definida pelo tamanho dos dados que serão processados ou do número de processadores no sistema. Com isso, para distinção do endereçamento, CUDA define algumas variáveis de sistema: *threadIdx* (índice da *thread*), *blockIdx* (índice do bloco), *blockDim* (dimensão dos blocos), *gridDim* (dimensão do *grid*). A Figura 2.5 apresenta um exemplo de um *grid* CUDA com três dimensões.

Em CUDA os programadores não precisam se preocupar com a criação ou destruição das *threads*, devem apenas especificar a dimensão do *grid* e dos blocos de *threads* necessários para o processamento de determinado *kernel*. Essas dimensões devem ser definidas ao realizar a chamada do *kernel*, no formato: `kernel <<< dimGrid,dimBlock >>> (argumentos)`. A sincronização das *threads* é realizada pela função `__syncthreads()`, que coordena a comunicação entre as *threads* de um mesmo bloco.

Modelo de Memória

As *threads* de CUDA acessam os dados a partir de vários espaços de memória durante a execução. Cada *thread* possui uma memória local privada. Enquanto isso, cada bloco de *threads* possui uma memória compartilhada visível a todas as *threads* pertencentes ao bloco. Além disso, todas as *threads* tem acesso a mesma memória global.

Programação Heterogênea

O modelo de programação CUDA assume que as *threads* executam em um dispositivo "device" fisicamente separado que opera como um coprocessador para o dis-

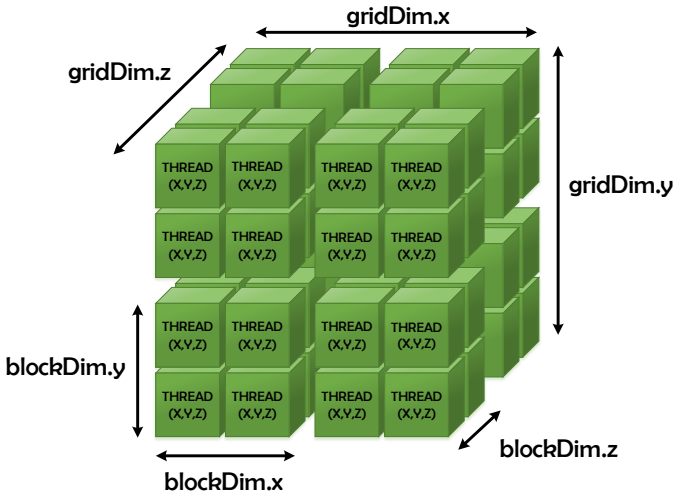


Figura 2.5: *Grid* tridimensional de CUDA *threads*.

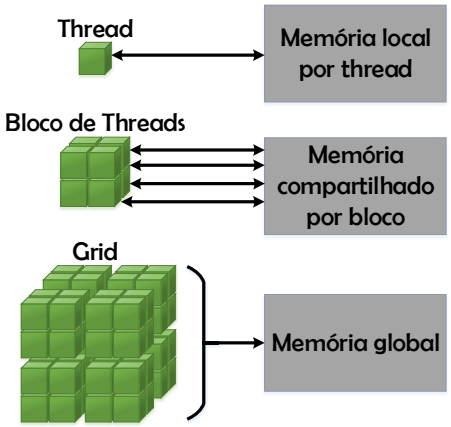


Figura 2.6: Hierarquia de Memória CUDA [NVIDIA, 2014b].

positivo "host" onde o programa C está rodando. Neste caso, os *kernels* executam em GPU e o resto do programa C executa em CPU. Esse modelo de programação também considera que tanto o *host* quanto o *device* mantém seus próprios espaços de memória separados na DRAM, chamadas de memória *host* e memória *device*, respectivamente.

Fluxo de Dados

A implementação de qualquer algoritmo em CUDA consiste em copiar os dados para GPU, realizar o processamento, e copiar os resultados de volta para CPU. Esse

tempo de cópia, juntamente com a chamadas dos *kernels* e a operação de sincronização, representam normalmente o gargalo dos algoritmos implementados em CUDA. Dessa forma, é importante analisar esse fluxo de dados e seu impacto nas implementações em GPU.

2.5 Métricas de Desempenho

Métricas de desempenho permitem avaliar o desempenho da aplicação paralela tendo por base o tamanho da entrada e espaço de memória, bem como o número de processadores e parâmetros específicos de comunicação da arquitetura, todos fatores que influenciam no *tempo de execução*. A seguir, são apresentadas as métricas de desempenho mais utilizadas em aplicações paralelas.

2.5.0.1 *Speedup*

O *Speedup* é definido como a razão entre o tempo de execução do algoritmo de forma sequencial e o tempo de execução do mesmo algoritmo utilizando vários processadores:

$$S_p = \frac{T_1}{T_p} \tag{2.2}$$

em que p é o número de processadores; T_1 é o tempo de execução do algoritmo sequencial; e T_p é o tempo de execução do algoritmo paralelo em p processadores.

Em comparação com o número de processadores, o *speedup* pode ser classificado da seguinte forma [Souza, 2013]:

- $S_p = p$, *Linear speedup*
- $S_p < p$, *Sub-linear speedup*
- $S_p > p$, *Super-linear speedup*

As leis de Amdahl e de Gustafson-Barsis apresentam limites de *speedup* em relação à quantidade de processadores e à parte do código que pode ser paralelizada e são descritas a seguir.

Lei de Amdahl A Lei de Amdahl [Amdahl, 1967] especifica o máximo *speedup* que pode ser esperado na paralelização de um programa sequencial. Essencialmente, esta lei diz que o máximo *speedup* S de um programa é

$$S = \frac{1}{(1-P) + \frac{P}{p}}, \tag{2.3}$$

onde P é a fração do tempo total de execução sequencial calculada pela parte do código que pode ser paralelizado e p é um número de processadores que executam a parte paralela do código. Essa lei mostra que o *speedup* esperado é maior quando uma grande parte do código pode ser paralelizada. Além disso, se P é um número pequeno, apenas uma pequena parte do código pode ser paralelizada, então aumentar a quantidade de p de processadores resulta em pequenas melhorias no desempenho do algoritmo paralelo. Dessa forma, para atingir o maior *speedup* para um problema de tamanho fixo, é importante aumentar o valor de P, maximizando a quantidade de código que pode ser paralelizada [NVIDIA, 2014a].

Lei de Gustafson-Barsis A Lei de Gustafson-Barsis mostra que, na prática, o tamanho de um problema se escala com a quantidade de processadores. Dessa forma, o maior *speedup* S de um programa é:

$$S = p + (1-P)(1-p), \tag{2.4}$$

2.5.0.2 Quadros Processados por Segundo

Esta métrica é utilizada para medir a eficiência de algoritmos paralelos de processamento de vídeos. Consiste em medir a quantidade de quadros do vídeo que foi processada por segundo:

$$Q = \frac{N}{T}, \tag{2.5}$$

em que Q é a quantidade de quadros processados por segundo; N é a quantidade total de quadros do vídeo; e T é o tempo de execução do algoritmo em segundos.

Capítulo 3

Revisão da Literatura

O foco deste trabalho é a criação e implementação de métodos paralelos de sumarização automática de vídeos, no entanto, não foram encontrados trabalhos desse tipo na literatura. Com isso, a seguir serão apresentadas diversas abordagens sequenciais de sumarização de vídeos, juntamente com algumas abordagens paralelas nas áreas de Visão Computacional e Processamento de Imagens e Vídeos, que podem estar relacionadas com este trabalho de alguma forma.

Mundur et al. [2006] apresentam um método baseado na triangulação de Delaunay para agrupar os quadros similares do vídeo. Os quadros do vídeo são descritos através do histograma de cor no espaço HSV. Cada quadro é representado por um vetor de características com 256 dimensões e a sequência do vídeo é representada por uma matriz. Para reduzir a dimensão da matriz é aplicada a técnica de análise dos componentes principais (PCA), que gera uma nova matriz de dimensão $n \times d$, onde n é número total de quadros selecionados e d é o número de componentes principais. Assim, cada quadro é representado por um vetor d-dimensional e o diagrama de Delaunay é construído para n pontos em d-dimensões. Os grupos são obtidos de acordo com as arestas de separação no diagrama. A remoção destas arestas identifica os grupos no diagrama. Para cada grupo, o quadro-chave é representado pelo quadro mais próximo do centróide. Os resumos gerados são comparados com os resumos do Open Video Project (OV) [OpenVideo, 2011].

Furini et al. [2007] propõem uma abordagem, denominada VISTO (*VIsual STOryboard*), que aplica um algoritmo de clusterização para selecionar os quadros mais representativos. Os quadros do vídeo também são descritos através do histograma de cor no espaço HSV. Cada quadro é representado por um vetor de características com 256 dimensões. O algoritmo *furthest-point-first* [Geraci et al., 2006] é utilizado para obter os grupos. Para determinar o número de agrupamentos é calcu-

lada a distância par-a-par entre os quadros. Se a distância for maior que um limiar, então o número de grupos é incrementado. Por fim, os quadros redundantes e inexpressivos (por exemplo, um quadro todo preto) são eliminados através da análise dos histogramas HSV. Os resumos gerados pelo método proposto são comparados com os resumos gerados em Mundur et al. [2006] e com os resumos do OV.

Avila et al. [2011] propõem uma abordagem chamada VSUMM para sumarização automática de vídeos baseada na extração de características de cor e no algoritmo de clusterização k-means. Inicialmente, o vídeo é segmentado em quadros e uma subamostra dos quadros é utilizada. Então são extraídas as características visuais presentes nos quadros, que serão utilizadas para a descrição do conteúdo do vídeo. Em seguida, os quadros são agrupados por meio de uma abordagem de classificação não-supervisionada (k-means). Para gerar o resumo estático, os quadros-chave semelhantes são eliminados com o propósito de manter a qualidade do resumo. Por fim, os quadros-chave são dispostos em ordem cronológica para facilitar o entendimento visual do resultado. Para avaliação dos resumos, foram utilizados os vídeos do OV e foi criada uma base de dados de vídeos com vídeos coletados na internet. Os resumos gerados foram comparados com os resumos gerados pelos métodos presentes em Mundur et al. [2006] e Furini et al. [2007].

Almeida et al. [2012] propõem uma abordagem de sumarização de vídeos chamada VISON (*VIdeo Summarization for ONline applications*). Esse método trabalha com o vídeo em seu domínio comprimido, sem realizar a decodificação. Características visuais, neste caso, histograma de cor no espaço HSV, são extraídas da *stream* do vídeo. Os quadros são agrupados de acordo com a métrica *Zero-mean Normalized Cross Correlation (ZNCC)*, utilizada para medir a distância entre dois quadros. O sumário gerado é filtrado comparando *pixels* nas imagens quantizadas. Os resumos gerados por esse trabalho foram comparados com os resumos gerados em Mundur et al. [2006]; Furini et al. [2007]; Avila et al. [2011] e com os resumos do Open Video Project.

Ejaz & Baik [2012] apresentam um método para extração dos quadros-chave de vídeos levando em consideração a visão do usuário em relação os quadros escolhidos. As características extraídas são o histograma de cor HSV e a textura através da matriz de co-ocorrências de níveis de cinza. Os vetores de características são agrupados de acordo com o algoritmo k-means. Neste método, a métrica de distância utilizada pelo algoritmo de agrupamento é baseada na função de base radial (RBF). O usuário faz parte do ajuste dos parâmetros da métrica RBF. Por fim os quadros que não possuem informações visuais úteis são retirados do resumo na etapa de pós-processamento. Os resumos gerados por esse trabalho foram comparados com

os resumos gerados em Mundur et al. [2006]; Furini et al. [2007]; Avila et al. [2011] e com os resumos do Open Video Project.

Em Kuanar et al. [2013] é proposto um método automático para extração de quadros-chave, onde o agrupamento é baseado na triangulação de Delaunay. Inicialmente, é realizada uma pré-amostragem dos quadros com o objetivo de dividir o vídeo em quadros que possuem informações significantes. Para isso é analisado um quadro por segundo e é aplicada a pré-amostragem baseada em informação mútua, que utiliza o cálculo da entropia *joint* entre quadros sucessivos. A seguir, os quadros do vídeo são descritos por características globais como cor e textura. Então, os quadros inexpressivos são encontrados a partir da variância dos histogramas e eliminados. Os quadros restantes são agrupados através de um grafo de Delaunay e uma estratégia iterativa de poda é aplicada para extração dos quadros-chave. Os resumos gerados por esse trabalho foram comparados com os resumos gerados em Mundur et al. [2006]; Furini et al. [2007]; Avila et al. [2011] e com os resumos do Open Video Project.

Cahuina et al. [2013] apresentam um método de sumarização de vídeos baseado na segmentação temporal do vídeo e em palavras visuais obtidas através de descritores locais. Inicialmente é realizada a segmentação temporal do vídeo, onde são detectados vários segmentos do vídeo com informações consideradas importantes. Para encontrar os quadros mais representativos de cada segmento, um algoritmo de clusterização é utilizado, selecionando os quadros mais próximos aos centróides como os representativos. Então são utilizados descritores locais para detecção e descrição de características nos quadros representativos extraídos anteriormente, juntamente a uma abordagem de *Bag-of-Words* (BoW), gerando o vocabulário de palavras visuais. Em seguida, são criados histogramas de ocorrência das palavras visuais do vocabulário para cada um dos quadros extraídos e os quadros representativos são agrupados de acordo com os histogramas de ocorrência de palavras visuais, ou seja, quadros com entidades visuais similares são selecionados para os mesmos grupos. Finalmente, o método filtra os resultados para eliminar possíveis quadros-chave duplicados. Os resumos gerados por esse trabalho foram comparados com os resumos gerados em Mundur et al. [2006]; Furini et al. [2007]; Avila et al. [2011] e com os resumos do Open Video Project.

Dentre os vários métodos de sumarização automática de vídeos presentes na literatura, os métodos apresentados anteriormente foram escolhidos de acordo com a relevância e com a possibilidade de comparação entre as abordagens (utilizaram as mesmas base de dados nos experimentos). A Tabela 3.1 apresenta um resumo comparativo entre as etapas desses métodos.

Tabela 3.1: Resumo comparativo entre alguns métodos de sumarização de vídeos.

Artigo	Segmentação do Vídeo	Extração de Características	Clusterização	Base de dados	Comparado em outro artigos?
Mundur et al. [2006]	Amostragem dos quadros	Histograma de cor	Diagrama de Delaunay	OV	Sim
Furini et al. [2007]	Amostragem dos quadros	Histograma de cor	FPF com heurísticas	OV	Sim
Avila et al. [2011]	Amostragem dos quadros	Histograma de cor	k-means, dist. Euclidiana	OV e VSUMM	Sim
Almeida et al. [2012]	Amostragem dos quadros	Histograma de cor	ZNCC	OV e VSUMM	Não
Ejaz & Baik [2012]	Amostragem dos quadros	Histograma de cor e textura	k-means, RBF	OV e VSUMM	Não
Kuanar et al. [2013]	Amostragem dos quadros	Histograma de cor e textura	Diagrama de Delaunay	OV e VSUMM	Não
Cahuina et al. [2013]	Temporal (em tomadas)	Descritores locais (HoG)	x-means e LBG	OV e VSUMM	Não

A escolha dos dois métodos de sumarização de vídeos utilizados como base para este trabalho [Avila et al., 2011; Cahuina et al., 2013] pode ser justificada através da análise da Tabela 3.1. As abordagens presentes em Mundur et al. [2006]; Furini et al. [2007]; Avila et al. [2011] são consideradas estado da arte em relação à sumarização automática de vídeos e são usadas amplamente para comparações com novos métodos. Devido a este fato, juntamente com os resultados superiores em relação aos outros dois métodos, o método presente em Avila et al. [2011] foi selecionado como base para este trabalho. Por outro lado, devido à grande diferença em relação aos outros métodos na questão da segmentação do vídeo e no tipo de característica extraída, e por apresentar comparações com os três artigos do estado da arte, a abordagem proposta por Cahuina et al. [2013] também foi escolhida como base para o atual trabalho. Outros artigos pesquisados em que os métodos propostos para sumarização de vídeos utilizam segmentação temporal e descritores locais não foram comparados com os métodos do estado da arte em seus experimentos. Outro ponto que influenciou a escolha dos dois métodos foi a acessibilidade ao autores.

Os artigos descritos a seguir mostram que existem diversos trabalhos atuais que resolvem problemas de processamento de imagens/vídeos e visão computacional através de arquiteturas paralelas, tanto através de GPUs quanto de vários processadores da CPU. Estes métodos servem como inspiração para este trabalho, que visa resolver o problema de sumarização automática de vídeos através de CPUs *multicore* e GPUs.

Sinha et al. [2011] descrevem implementações dos algoritmos de rastreamento e extração de características, KLT [Tomasi & Kanade, 1991] e SIFT [Lowe, 1999], em unidades de processamento gráfico (GPUs). Em ambos os casos, a estratégia desenvolvida foi dividir a computação entre CPU e GPU na melhor forma possível, respeitando as restrições do modelo de computação em GPU. A implementação do KLT conseguiu melhoria do tempo de execução em relação à implementação em CPU de aproximadamente 20 vezes, enquanto o SIFT em GPU foi 10 vezes mais rápido que a implementação sequencial.

Em Cheung et al. [2010] são apresentados os desafios e vantagens da implementação em GPU de algoritmos de codificação e decodificação de vídeos. Os autores analisam como os módulos destes algoritmos podem ser implementados de forma a expor o maior paralelismo possível, resultando na utilização total da capacidade paralela das GPUs. Além disso, é apresentado um exemplo de algoritmo de decodificação que particiona o fluxo entre CPUs e GPUs, atingindo o máximo de computações simultâneas. Os resultados mostraram que as implementações em GPU podem atingir um *speedup* considerável para os módulos de computação intensiva do algoritmo de codificação.

Em Clemons et al. [2011], é apresentada uma arquitetura heterogênea *multicore*, chamada EFFEX, que utiliza funções e memória otimizadas para reduzir significativamente o tempo de execução de algoritmos de extração de características em plataformas móveis. Para criação desta arquitetura foram analisadas as características de execução de três algoritmos comuns de extração de características: FAST, HoG e SIFT. Os resultados experimentais apresentaram *speedup* de aproximadamente 14× para extração de características com uma redução, em relação à energia, de 40× para acessos de memória.

Wu et al. [2011] apresentam o projeto e implementação de algoritmos de *Bundle Adjustment* que exploram paralelismo em *hardware* para solução eficiente de problemas de larga escala para reconstrução de cenas 3D. O *Bundle Adjustment* foi implementado em *multicore* CPUs, apresentando *speedup* de 10×, e GPUs, atingindo *speedup* de até 30× em relação aos métodos do estado da arte, reduzindo também a quantidade de memória utilizada.

Li et al. [2012] apresentam uma abordagem interativa, utilizando GPUs, para aplicar uma nova textura sobre imagens/vídeos (*retexturing*), mantendo as características da textura original, como a distorção de textura subjacente. O sistema oferece uma interação em tempo real onde os usuários podem selecionar os objetos que obterão a nova textura, como será esta nova textura, entre outros ajustes. Essa abordagem utiliza características SIFT para descobrir naturalmente a distorção de

textura que deve ser mantida. A interatividade em tempo real é facilitada por um *grid* bilateral em GPU. Os experimentos atingiram resultados com alta qualidade do *retexturing* e rápidos (processando até 80 quadros por segundo) devido ao uso de GPUs.

Holub et al. [2013] propõem algoritmos paralelos e eficientes para codificação e decodificação JPEG em GPU, com foco em aplicações interativas e em tempo real. O objetivo principal é acelerar transmissões de vídeos em alta resolução como HD, 2K e 4K. Os autores analisaram vários aspectos dos algoritmos propostos incluindo a dependência em relação ao conteúdo da imagem e em várias configurações do algoritmo de compressão. Os experimentos demonstram que esses algoritmos podem ser executados de forma eficiente para vídeos de resolução até 4K, com pequeno impacto na latência.

Farias et al. [2013] introduzem um algoritmo de segmentação de imagens baseado em grafos, chamado *Reduction Sweep*, desenvolvido de forma paralela. Esse algoritmo utiliza características locais das imagens, o que permite com que cada *pixel* seja comparado com seus vizinhos de maneira independente, e os *pixels* similares de acordo com um critério de cor são unidos. Esta natureza independente desse algoritmo gerou um método rápido, mantendo a qualidade visual dos resultados. São apresentadas quatro implementações diferentes do algoritmo: CPU sequencial, CPU paralela, GPU e híbrida CPU-GPU.

Pedronette et al. [2013] propõem uma solução paralela utilizando GPUs para o problema de *re-ranking*, utilizado para recuperação de imagens baseada no conteúdo. Os resultados demonstraram um desempenho significativo com a implementação em GPU, atingindo *speedup* de 7$\times$ em relação à versão sequencial e até 36$\times$ em partes do algoritmo.

Capítulo 4

Implementações Propostas

Este trabalho propõe a paralelização de dois algoritmos de sumarização automática de vídeos, descritos em Avila et al. [2011] e Cahuina et al. [2013]. As etapas destes métodos foram descritas nas Subseções 2.1.1 e 2.1.2, respectivamente, e a seguir são apresentadas as implementações paralelas de cada uma destas etapas. Com o objetivo de acelerar ao máximo o desempenho de ambos algoritmos, foram implementadas três versões para cada algoritmo. A primeira versão utiliza CPU com múltiplos núcleos, a segunda utiliza GPUs e, por fim, a última versão, chamada híbrida, apresenta uma combinação das etapas mais eficientes das versões anteriores, buscando maximizar o desempenho. Como a versão híbrida necessita dos resultados de tempo de execução das versões *multicore* CPU e GPU, ela é apresentada na Seção 5, após os experimentos com estas outras versões. Todas as implementações estão disponíveis na Internet [Almeida, 2014].

É importante ressaltar que as implementações paralelas são fiéis às originais, utilizando os mesmos algoritmos e as mesmas etapas. No Capítulo 6 é descrito como trabalho futuro realizar modificações nos métodos originais com o objetivo de melhorar os resultados das implementações paralelas.

4.1 Implementações Paralelas para Sumarização proposta em Avila et al. [2011]

A seguir são descritas as implementações paralelas (*multicore* CPUs e GPUs) para a abordagem de sumarização automática de vídeos proposta em Avila et al. [2011]. A Figura 4.1 apresenta as etapas do método e o fluxo de dados em cada etapa para as implementações em GPUs e *multicore* CPUs. Como é possível ver nessa figura,

as duas últimas etapas do VSUMM não foram implementadas de forma paralela pois, segundo nossa análise, apresentam tempo de execução insignificante com o algoritmo sequencial.

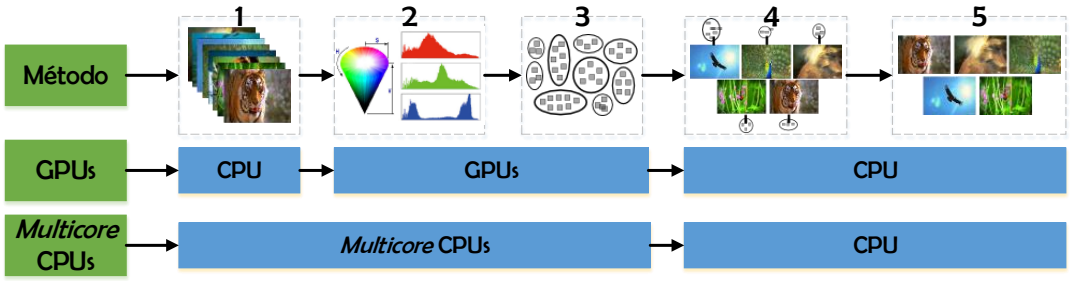


Figura 4.1: Diagrama de Fluxo de Dados/Execução das Implementações Paralelas para a Abordagem de Sumarização de Vídeos proposta em Avila et al. [2011]

4.1.1 Versão CPU multicore

Etapa 1 - Segmentação/Decodificação do Vídeo

A segmentação do vídeo em quadros sob uma amostragem de um quadro por segundo em CPU *multicore* é realizada pela biblioteca OpenCV. Uma grande vantagem das CPUs *multicore* sobre as GPUs é a possibilidade das *threads* executarem funções de diversas bibliotecas, como a OpenCV. Para paralelizar esta etapa, cada *thread* possui a tarefa de decodificar uma parte do vídeo e a quantidade de *threads* é proporcional à quantidade de núcleos da CPU. A definição de qual parte do vídeo será segmentada por qual é *thread* é realizada dividindo a quantidade total de quadros pela quantidade de *threads* disponíveis. Desta forma, cada *thread* sabe os índices dos quadros que terá que decodificar. A biblioteca OpenCV permite o acesso a um quadro específico do vídeo e sua decodificação, o que possibilita a decodificação de partes distintas do vídeo simultaneamente. As imagens decodificadas são salvas em disco para compor o resumo final.

Etapa 2 - Extração de Características e Eliminação de Quadros Inexpressivos

A extração do histograma de cor dos quadros e a conversão das imagens do espaço de cor RGB para o HSV nesta versão *multicore* consistem em dividir os quadros entre as *threads*, de forma que cada *thread* realize essas operações nos quadros que foram designados a ela. Cada *thread*, antes da conversão dos quadros para o espaço HSV e do cálculo do histograma, realiza a eliminação dos quadros inexpressivos. Ou seja,

as *threads* inicialmente calculam o histograma RGB e o desvio padrão do mesmo. Caso a análise do desvio padrão conclua que o quadro é inexpressivo, a *thread* passa a analisar o próximo quadro. Caso contrário a *thread* realiza as operações de extração de características no quadro.

Etapa 3 - Agrupamento dos Quadros

Como apresentado anteriormente, o algoritmo de agrupamento utilizado neste método é o *k-means*. Duas etapas do *k-means* são mais custosas computacionalmente e podem ser paralelizadas: o cálculo do centroide de cada grupo e a busca do grupo mais próximo de cada quadro. Como estas tarefas são executadas várias vezes até o algoritmo convergir, é interessante a aceleração das mesmas. Para isso, os quadros são divididos entre as *threads* e elas encontram o grupo mais próximo destes quadros simultaneamente. Para encontrar o novo centroide de cada grupo, é necessário calcular a média dos membros dos grupos. Como as *threads* podem atribuir quadros ao mesmo grupo paralelamente, apenas o somatório dos elementos de cada grupo é realizado por elas. O somatório dos elementos de cada grupo é realizado localmente por cada *thread* para evitar que mais de uma *thread* realize a soma simultaneamente, causando conflito. Ao final, as somas locais são unidas para formar o somatório global de cada grupo. Ao fim da iteração, este somatório é dividido pela quantidade de quadros que foram atribuídos a cada grupo, encontrando assim os centroides. E este processo é repetido até a convergência do algoritmo de agrupamento.

4.1.2 Versão GPU

Etapa 1 - Segmentação/Decodificação do Vídeo

Neste algoritmo a segmentação do vídeo consiste na decodificação do mesmo, extraindo uma amostragem de um quadro por segundo. A NVIDIA possui a biblioteca NVCUVID [NVIDIA, 2014c], que tem como objetivo a decodificação de vídeos. A implementação em CPU desta etapa, salva os quadros no disco rígido para posterior geração do resumo final, sem a necessidade de decodificar o vídeo novamente. No entanto, decodificar o vídeo em GPU e salvá-lo no disco rígido é mais custoso, pois é necessário copiar o quadro da GPU para CPU, salvá-lo, e depois copiá-lo de volta para GPU para os próximos processamentos.

Com isso, foram implementadas três versões de segmentação do vídeo. A primeira consiste em decodificar o vídeo, utilizando a biblioteca NVCUVID, de forma que um quadro é extraído e mantido na GPU para posterior processamento. Ao

encerrar o processamento, o quadro é copiado para CPU, salvo no disco, e o próximo quadro é decodificado. A segunda implementação realiza a decodificação em GPU, copia os quadros para CPU e salva-os. Já a terceira versão é a utilização da segmentação em CPU. Esta última escolha foi realizada com o objetivo de comparar o gargalo de cópia em CPU e GPU com as outras duas implementações da segmentação.

Etapa 2 - Extração de Características e Eliminação de Quadros Inexpressivos

Foram implementadas duas formas de extração de características e eliminação de quadros inexpressivos: 1) computar um quadro por vez; 2) computar a maior quantidade de quadros que cabem na GPU de uma só vez. Ambas formas implementadas são descritas a seguir.

Um quadro por vez: Esta extração foi elaborada para utilização com a primeira implementação da decodificação, onde cada quadro é decodificado em GPU e mantido nela para realização desta etapa. Dessa forma, a extração de características é executada para um quadro por vez. A primeira parte da extração consiste em transformar os quadros do espaço de cor RGB para HSV. Para isso, cada *thread* do *grid* acessa o valor RGB de um *pixel* da imagem e calcula o valor HSV correspondente do *pixel*.

A segunda parte da extração é responsável pelo cálculo de histograma de cor. Calcular histogramas em CPU é trivial e fácil de implementar. Porém em GPU, para obter o resultado com eficiência, o cálculo não é tão simples. O problema é que múltiplas *threads* podem querer incrementar a mesma faixa (bin) do histograma ao mesmo tempo, o que gera conflito entre as *threads* (ilustrado na Figura 4.2). Para resolver isso, são utilizados incrementos atômicos, que forçam o algoritmo a sequenciar os incrementos. Funções atômicas solucionam problemas de sincronização e coordenação em sistemas de computação paralela. O conceito geral é prover um mecanismo para uma *thread* atualizar uma localização de memória de forma que esta atualização seja atômica (sem interrupção), respeitando as outras *threads*. Em CUDA, a função `atomicAdd(addr,y)` gera uma sequência de operações atômicas que consistem na leitura do valor no endereço `addr`, na soma de `y` a este valor e no armazenamento do resultado de volta no endereço de memória `addr`.

Dessa forma, o cálculo do histograma em GPU é dividido em duas fases. Na primeira fase, cada bloco computa um histograma dos dados que suas *threads* examinam. Como cada bloco faz isso independentemente, esses histogramas podem ser computados na memória compartilhada, economizando o tempo. Esta operação

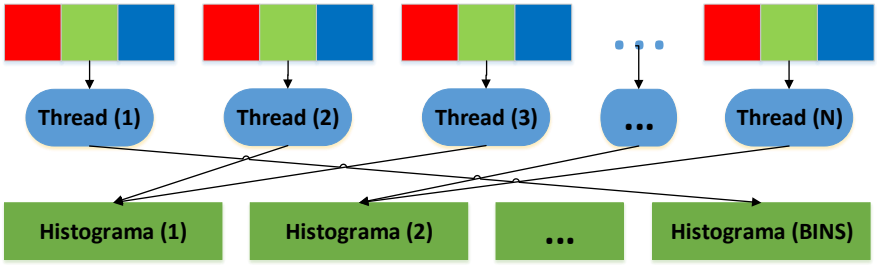


Figura 4.2: Computação paralela em GPU de um histograma com BINS faixas e N *threads*. Atualizações no histograma geram conflitos e requerem sincronização das *threads* ou atualizações atômicas do histograma na memória.

não acaba com a necessidade de operações atômicas, pois múltiplas threads dentro do bloco podem examinar elementos com o mesmo valor. Como é utilizada a memória compartilhada e ela possui limitações de tamanho, a quantidade de threads responsáveis pelo cálculo foi definida como 256. Além disso, a quantidade de faixas do histograma também foi definida como 256 pois, desta forma, o fato de 256 threads estarem competindo pelos 256 endereços do histograma reduz a contenção que poderia ser gerada se milhares de threads estivessem competindo por 256 endereços ou se 256 *threads* estivessem competindo por uma pequena quantidade de endereços. A segunda fase consiste na junção de cada histograma temporário calculado nos blocos em um histograma global. Esta tarefa envolve adicionar cada entrada dos histogramas dos blocos à entrada correspondente no histograma final, como ilustrado na Figura 4.3. Como o histograma possui 256 faixas e são usadas 256 *threads*, cada *thread* soma uma única faixa ao histograma final. Note que não se tem nenhuma garantia sobre qual ordem os blocos somam seus valores ao histograma final, mas como a soma de inteiros é comutativa, o resultado sempre será o mesmo [Sanders & Kandrot, 2010].

Todos os quadros que passaram por este processo são considerados quadros com informações relevantes, pois os quadros inexpressivos são eliminados em um passo anterior. Para essa eliminação, o histograma RGB de cada quadro é calculado na mesma forma apresentada acima e então é calculado o desvio padrão desse histograma, definido de acordo com a fórmula $stdev = \sqrt{\frac{1}{n-1}\sum_{i=1}^{n}(x_i - \overline{x})^2}$, em que n é o tamanho do histograma e x_i representa cada faixa e $\overline{x}$ representa a média do histograma. Como o histograma será normalizado, não é necessário computar a média, pois com o histograma normalizado, a média passa a ser $\frac{1}{n}$. Dessa forma, cada CUDA *thread* normaliza e realiza a computação $(x_i - \overline{x})^2$ de uma faixa do histograma. Então as *threads* são sincronizadas e é realizada uma operação de redução,

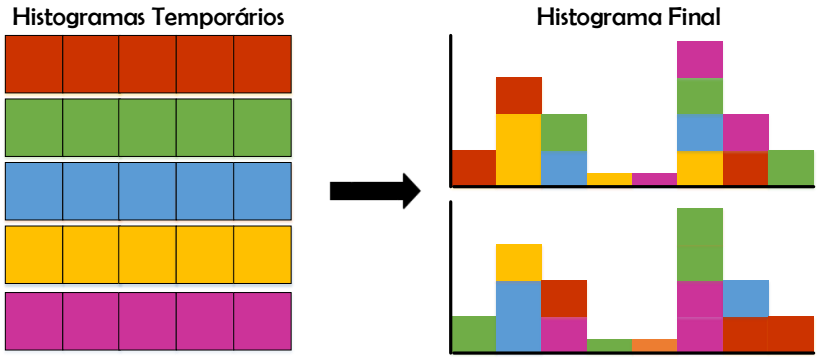


Figura 4.3: Ilustração do Cálculo de Histograma em GPU

responsável pelo somatório dos valores calculados. Finalmente, o valor obtido do somatório é dividido por $\frac{1}{n-1}$ e é extraída a raiz quadrada, encontrando assim o desvio padrão, e utilizando este valor para eliminar os quadros inexpressivos.

Vários quadros por vez: Esta versão foi utilizada com a segunda e terceira formas de segmentação do vídeo, após salvar todos os quadros, estes são carregados para GPU até que a memória da GPU esteja completa e então é realizada a extração de características. Com o objetivo de realizar a extração de características para a maior quantidade de quadros possível, o *grid* do CUDA foi considerado como um *grid* tridimensional. As duas primeiras dimensões são utilizadas como a largura e altura dos quadros, enquanto a terceira dimensão é utilizada para a quantidade de quadros que foram copiados para GPU, exemplificadas na Figura 4.4.

A transformação da imagem de RGB para HSV é realizada similar à forma anterior de extração de características. Cada *thread* é responsável pelo cálculo do valor do *pixel* RGB correspondente em HSV. A única diferença é que na maneira anterior, o cálculo era realizado simultaneamente para apenas um quadro, e nesta maneira o cálculo é realizado para muitos quadros em paralelo. Ou seja, o *grid* bidimensional era suficiente para a extração de características de apenas um quadro, enquanto neste caso é necessário o *grid* tridimensional.

Novamente, após a transformação do espaço de cor, o histograma de cada quadro é computado. Como citado anteriormente, computar histogramas em GPU não é trivial e uma forma de solucionar o problema de conflito entre as *threads* para atualizar algum valor do histograma é a utilização de incrementos atômicos. Diferentemente da versão anterior, como vários quadros são copiados para GPU simultaneamente, não foi utilizada a memória compartilhada para o cálculo do histograma.

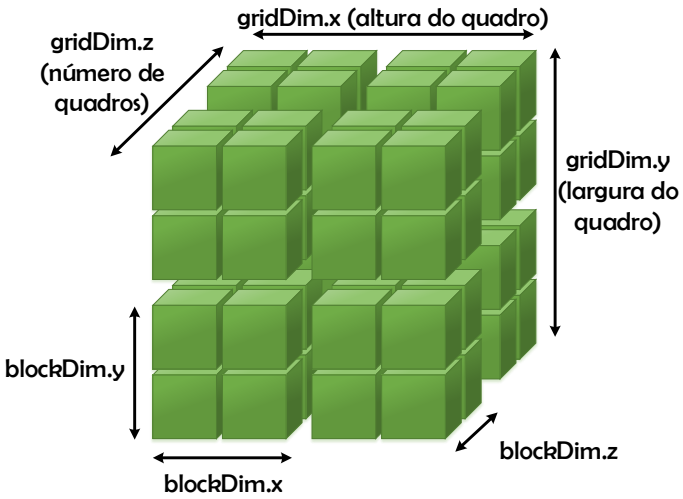


Figura 4.4: CUDA *grid* tridimensional com informações de dimensões utilizadas para extração de características de vários quadros simultaneamente.

Isto ocorreu devido a dois motivos. Primeiro, a memória compartilhada tem tamanho limitado, limitando assim sua utilização para muitos dados. Segundo, não há garantia que todas as *threads* do mesmo bloco vão acessar os *pixels* da mesma imagem para incremento, o que pode acarretar em cálculos errados. Dessa forma, cada *thread* do *grid* tridimensional acessa um *pixel* de um dos quadros e realiza o incremento atômico no histograma final. Ao final deste processo, um vetor com todos os histogramas é gerado.

Como na implementação anterior, a eliminação de quadros inexpressivos é realizada antes da extração de características. Para isso, os histogramas RGB dos quadros são calculados da mesma forma explicada acima. Com os histogramas na GPU, cada *thread* de um *grid* bidimensional acessam um valor de histograma, normalizam este valor e computam o quadrado deste valor subtraído pela média dos histograma. Ao final, estes valores são somados e é extraída da raiz quadrada, encontrando assim o desvio padrão.

Etapa 3 - Agrupamento dos Quadros

Nesta versão, o cálculo do grupo mais próximo dos quadros é realizado de forma paralela em GPU. Para isso, cada CUDA *thread* encontra o grupo mais próximo de cada quadro utilizando a distância Euclidiana. A seguir, as *threads* são sincronizadas para o cálculo de quantos quadros mudaram de grupo. Esta contagem é necessária pois quando os quadros param de se mover para outros grupos, o algoritmo de

clusterização convergiu. Então a informação de qual grupo cada quadro pertence é copiada para CPU e os novos centroides são calculados. Estes passos são repetidos até a convergência do *k-means*.

4.2 Implementações Paralelas para Sumarização proposta em Cahuina et al. [2013]

A seguir são descritas as implementações paralelas (multicore CPUs e GPUs) para a abordagem de sumarização automática de vídeos proposta em Cahuina et al. [2013]. A Figura 4.5 apresenta as etapas do método juntamente com o fluxo de dados das suas implementações paralelas. A partir da figura é possível concluir que na implementação utilizando GPUs ocorreu um grande fluxo de dados entre CPU e GPU, onde as etapas 1, 3, 6 e 7 foram implementadas sequencialmente em CPU e as etapas 2, 4 e 5 foram implementadas de forma paralela em GPU. Já a implementação utilizando *multicore* CPU, as seis primeiras etapas foram executadas utilizados os vários processadores do CPU e apenas a última etapa é sequencial, utilizando um processador.

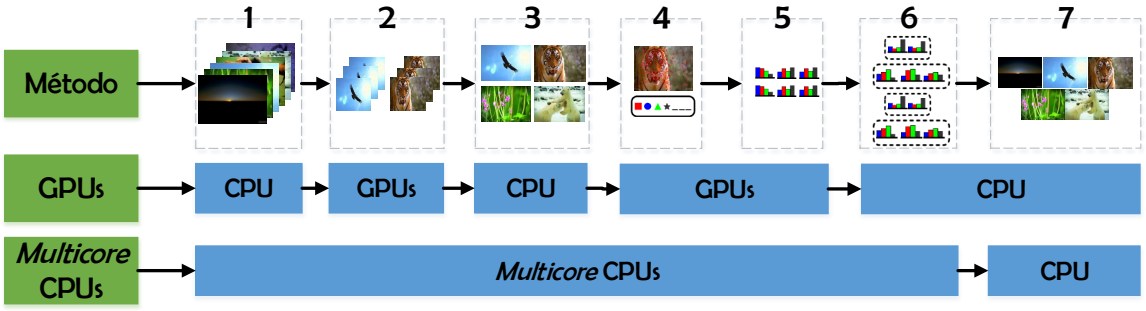


Figura 4.5: Diagrama de Fluxo de Dados/Execução das Implementações Paralelas para a Abordagem de Sumarização de Vídeos proposta em Cahuina et al. [2013]

4.2.1 Versão CPU *multicore*

Etapa 1 - Decodificação do Vídeo

A diferença desta decodificação do vídeo para a segmentação do vídeo apresentada no método VSUMM é apenas na quantidade de quadros que são extraídos. Neste caso, todos os quadros são utilizados. Então, a quantidade total de quadros do vídeo

é dividida entre as *threads*, e estas utilizam a biblioteca OpenCV para decodificar os vídeos e salvá-los no disco rígido.

Etapa 2 - Segmentação Temporal do Vídeo

A segmentação temporal do vídeo é realizada encontrando transições de corte e eliminando efeitos visuais como *fade-in/out*. Para essas tarefas, são computados os histogramas RGB dos quadros do vídeo, bem com a dissimilaridade de cossenos entre dois quadros consecutivos e a variância dos quadros. A realização desta etapa em CPU *multicore* consiste em dividir a quantidade total de quadros entre as *threads* disponíveis e cada *thread* fica responsável por estes cálculos simultaneamente. Uma vez que estes valores são calculados, o algoritmo prossegue a segmentação do vídeo em CPU pois as outras operações não são paralelizáveis ou já são eficiente em relação ao tempo de execução em CPU.

Etapa 3 - Detecção de Quadros Representativos

Esta detecção é realizada através da aplicação do algoritmo de clusterização *x-means* às tomadas extraídas na segmentação do vídeo. Desta forma, estas tomadas são divididas entre as *threads* disponíveis e cada uma delas, simultaneamente, aplica o algoritmo *x-means* em suas respectivas tomadas. Neste caso, foi utilizado um executável do *x-means*, fornecido pelos autores do método.

Etapa 4 - *Bag of Visual Words*

Cahuina et al. [2013] realizam vários experimentos com diversos descritores locais (SIFT, HoG, SURF) para a composição da detecção e descrição de características que compõe a abordagem de *Bag of Visual Words*. Segundo os autores, todos os descritores analisados apresentaram resultados similares, sem apresentar diferenças estatisticamente falando. Dessa forma, o descritor HoG foi escolhido para ser utilizado neste trabalho.

Para realizar esta tarefa paralelamente, as *threads* responsáveis por detectar os quadros representativos da etapa anterior, também são responsáveis pela detecção e descrição dos quadros através do HoG. Ou seja, várias *threads* simultâneas encontram quadros representativos com o algoritmo *x-means* e extraem as características locais destes quadros.

A computação das *codewords* é realizada pelo algoritmo de clusterização LBG, responsável por agrupar os vetores de características extraídos. Este algoritmo é similar ao *k-means*, explicado anteriormente, e, como no *k-means*, possui a etapa de

encontrar o grupo mais próximo de cada dado analisado (vetores de características). Esta etapa é paralelizável e foi implementada igual à mesma etapa do *k-means*, os dados são divididos entre as *threads*, que, simultaneamente realizam os cálculos de distância entre eles e os grupos e atribuem os dados ao grupo mais próximo. Os vetores de características mais próximos ao centroides de cada grupo são escolhidos como *codewords* e o vocabulário visual é gerado.

Etapa 5 - Histograma de *Visual Words*

Para contabilizar a ocorrência das palavras visuais escolhidas na etapa anterior presentes nos quadros representativos, os vetores de características de cada quadro são comparados com todas as palavras visuais escolhidas, e as faixas do histograma correspondentes às palavras visuais que ocorrem nos vetores são incrementadas. A comparação das características de cada quadro com as palavras visuais e a geração dos histogramas é realizada simultaneamente pelas *threads* disponíveis para os quadros representativos. Como os incrementos nas faixas do histograma podem gerar conflito entre as *threads*, estes são realizados de forma atômica.

4.2.2 Versão GPU

Etapa 1 - Decodificação do Vídeo

A decodificação do vídeo em GPU foi implementada da mesma forma que a segmentação do vídeo do método de sumarização apresentado anteriormente (Subseção 4.1.2), através de três tipos diferentes de decodificação. A única diferença é que no outro método foi utilizado um quadro por segundo, já nesta abordagem, todos os quadros foram extraídos.

Etapa 2 - Segmentação Temporal do Vídeo

Semelhantemente à etapa de extração de características do método de sumarização apresentado anteriormente, esta etapa foi implementada de duas formas seguindo os tipos de decodificação que foram realizados. A primeira forma consiste em calcular o histograma RGB e a variância para cada quadro decodificado de uma vez. E a outra forma realiza o cálculo do histograma RGB, variância e dissimilaridade de cossenos para todos os quadros que cabem na GPU simultaneamente.

Um quadro por vez: Esta etapa foi elaborada para utilização com a primeira implementação da decodificação, onde cada quadro é decodificado em GPU e mantido

nela para realização desta etapa. Dessa forma, o cálculo do histograma RGB e da variância são executados para um quadro por vez. Como uma imagem é uma matriz de *pixels*, o *grid* CUDA é bidimensional, uma dimensão para acessar as linhas e outra para as colunas da matriz.

O cálculo do histograma é realizado da mesma forma como foi explicado na Subseção 4.1.2. Inicialmente as *threads* computam o histograma dos *pixels* da imagem e cada bloco do *grid* armazena histogramas temporários de suas *threads* na memória compartilhada. A seguir, estes histogramas temporários são unificados formando o histograma final, de forma que cada *thread* soma uma única faixa ao histograma final.

Para o cálculo da variância de cada quadro, inicialmente a imagem é convertida para escala de cinza. Para isso, cada CUDA *thread* acessa um *pixel* RGB da imagem e realiza a conversão para escala de cinza. Agora, com cada quadro sendo uma matriz comum, as *threads* realizam o somatório de seus valores, são sincronizadas, e o somatório é dividido pela quantidade de valores da matriz, obtendo assim a média. A seguir, cada *thread* acessa um valor da matriz e computa este valor subtraído da média e elevado ao quadrado. As *threads* são sincronizadas novamente e outro somatório é realizado, encontrando assim a variância final.

O cálculo da dissimilaridade de cossenos utiliza dois quadros consecutivos, então não foi realizada em GPU, bem como as outras partes da segmentação.

Vários quadros por vez: Esta versão foi utilizada com a segunda e terceira formas de segmentação do vídeo, após salvar todos os quadros, estes são carregados para GPU até que a memória da GPU esteja completa e então são calculados o histograma RGB, variância e dissimilaridade de cossenos. Como são analisados vários quadros simultaneamente, o *grid* do CUDA utilizado foi o ilustrado na Figura 4.4. Novamente, o cálculo do histograma foi igual ao realizado no método VSUMM: cada *thread* do *grid* tridimensional acessa um *pixel* de um dos quadros e realiza o incremento atômico no histograma final. Ao final deste processo, um vetor com todos os histogramas é gerado.

Com todos os histogramas calculados, é possível computar a dissimilaridade de cossenos entre dois quadros consecutivos. Uma *thread* do CUDA é responsável por computar a dissimilaridade de dois histogramas consecutivos. Não há conflito entre as *threads* neste caso pois mesmo elas podendo acessar o mesmo histograma ao mesmo tempo, isto ocorre apenas para leitura. A escrita, ou seja, o resultado da dissimilaridade é salvo em uma posição diferente por cada *thread*.

Finalmente, para o cálculo das variâncias dos quadros, estes são convertidos

para escala de cinza. Seguindo o mesmo raciocínio anterior, cada *thread* acessa um *pixel* dos quadros e realiza a conversão. A próxima etapa para computar a variância é encontrar a média de cada matriz. Para isso, como várias matrizes estão na GPU, cada *thread* é responsável por encontrar a média de uma matriz, evitando conflitos entre as *threads*. A seguir, quando todas as médias foram calculadas, novamente, cada *thread* computa a variância de cada matriz. Note que, nas operações onde cada *thread* é responsável por uma matriz não há necessidade do *grid* tridimensional.

Etapa 3 - Detecção de Quadros Representativos

Esta detecção é realizada através da aplicação do algoritmo de clusterização *x-means* às tomadas extraídas na segmentação do vídeo. Devido à complexidade deste algoritmo de clusterização, em todas as versões foi utilizado o programa executável fornecido pelos autores. Dessa forma, esta etapa não foi implementada em GPU.

Etapa 4 - *Bag of Visual Words*

Existem várias implementações de descritores locais em GPU, e isto ocorre também com o descritor HoG escolhido para detecção e descrição de características. Desta forma, foi utilizada a implementação paralela disponibilizada na biblioteca OpenCV, implementada com base na versão original do HoG [Dalal & Triggs, 2005]. A biblioteca OpenCV também disponibiliza implementações em GPU para os descritores como SURF e FAST.

Por fim, a abordagem *Bag of Visual Words* gera o vocabulário de palavras visuais. Como descrito anteriormente, esta tarefa é realizada através de um algoritmo de agrupamento sobre os vetores de características, neste caso, o LBG foi utilizado. Este algoritmo é similar ao *k-means* e sua parte paralelizável consiste na tarefa de encontrar o grupo mais próximo de cada dado analisado (cada vetor de característica). Como na implementação em GPU desta etapa do *k-means*, cada CUDA *thread* encontra o grupo mais próximo de cada vetor de característica utilizando a distância Euclidiana. Um vetor guardando o grupo mais próximo de cada característica é atualizado pelas *threads* e copiado para CPU ao final do processo. Então os novos centroides são encontrados e esses passos repetem até o algoritmo convergir.

Etapa 5 - Histograma de *Visual Words*

Como na versão *multicore* do cálculo dos histogramas de *visual words*, a implementação paralela focou na comparação dos vetores de características de cada quadro

com todas as palavras visuais do vocabulário. Dessa forma, cada CUDA *thread* é responsável por gerar o histograma de ocorrência relacionado a cada quadro representativo. Para isso, as *threads* encontram as palavras visuais mais similares aos vetores de características e realizar os incrementos atômicos nas faixas correspondentes dos histogramas.

Capítulo 5

Experimentos e Resultados

Este capítulo apresenta diversos experimentos realizados, e suas respectivas análises, das implementações paralelas propostas. A partir dos resultados obtidos com as versões em GPU e *multicore* CPU, é proposta uma versão híbrida, com os melhores resultados atingidos para cada um dos métodos de sumarização analisados. Os experimentos com essa nova versão também são apresentados. Por fim, é realizada a análise de todos os resultados obtidos, bem como uma comparação entre os dois métodos de sumarização automática de vídeos em estudo. Os experimentos computaram os tempos de execução dos algoritmos. Também foram analisadas duas métricas de desempenho: *speedup* e quadros processados por segundo.

Os experimentos foram realizados em um computador composto pela CPU Intel Core i7 (3.40GHz), com 16GB de memória RAM, e GPU NVIDIA GeForce GTX 650 Ti (2GB de memória dedicada e 768 CUDA *cores*), com sistema operacional Windows 8.1. Este computador, com essas configurações, foi escolhido devido a sua disponibilidade para a execução dos experimentos.

Este capítulo está organizado na seguinte forma. A Seção 5.1 apresenta informações sobre o conjunto de vídeos utilizado nos experimentos. Nas Seções 5.2 e 5.3 são descritos e apresentados os experimentos realizados com as implementações paralelas das abordagens de sumarização de vídeos propostas em Avila et al. [2011] e em Cahuina et al. [2013], respectivamente. A análise dos resultados obtidos nessas seções é apresentada na Seção 5.4. Por fim, a Seção 5.5 apresenta uma discussão sobre uma implementação paralela genérica para a sumarização automática de vídeos.

5.1 Conjunto de Vídeos

Para análise da eficiência dos algoritmos paralelos propostos neste trabalho foi criado um conjunto de vídeos formado com 240 vídeos do YouTube, apresentados na Tabela 5.1. Foram escolhidos vídeos com quatro resoluções diferentes (320 × 240, 640 × 360, 1280 × 720 e 1920 × 1080) e seis tempos de duração (1, 3, 5, 10, 20 e 30 minutos). Os vídeos estão no formato AVI, os quadros por segundo variam entre 25 e 30, são coloridos e possuem áudio. Além disso, apresentam 10 diferentes gêneros como jornal, desenho, esportes, entre outros. Para cada combinação de resolução e duração, foram escolhidos 10 vídeos, tomando-se o cuidado de selecionar gêneros de vídeos diferentes.

A base de dados de vídeos Open Video OpenVideo [2011], muito utilizada para experimentos de métodos de sumarização de vídeos, não foi utilizada nos experimentos deste trabalho pois os vídeos são de baixa resolução e curta duração. Dessa forma, o conjunto de dados formados com vídeo do YouTube foi escolhido para ser utilizado nesses experimentos pois apresenta mais variedades de resolução, duração e gêneros.

5.2 Experimentos para a Abordagem VSUMM [Avila et al., 2011]

Os experimentos realizados foram divididos em três etapas. Inicialmente foram obtidos os tempos de execução da versão sequencial, necessários para comparações com as outras versões. A seguir foram avaliadas as versões paralelas *multicore* CPU e GPU. A versão *multicore* CPU foi implementada utilizando duas bibliotecas de algoritmos paralelos, OpenMP e *threads C++11*. Por outro lado, em GPU, foram implementadas as três versões apresentadas na Subseção 4.1.2, todas utilizando CUDA. Esses experimentos são apresentados a seguir.

5.2.1 Experimentos Versão Sequencial

A Tabela 5.2 apresenta os tempos médios de execução da sumarização automática de vídeos na versão sequencial. São apresentados os tempos das três primeiras etapas do VSUMM, segmentação do vídeo, extração de características e agrupamento dos quadros, seguidos pelo tempo total de execução do algoritmo. As duas últimas etapas foram ocultadas da tabela devido ao tempo de execução ser muito baixo, pró-

Tabela 5.1: Conjunto de vídeos Youtube (foram utilizadas as resoluções 320 × 240, 640 × 360, 1280 × 720 e 1920 × 1080 de cada vídeo)

Nome	Gênero	Duração (min)	# Quadros
Animal Planet - Wildest Season Advert 2013	Animais	1	1500
Animal Planet #1	Animais	3	5395
Awesome Animals	Animais	5	8992
Wings to Paradise	Animais	10	17983
Various Animals At The Miami Zoo	Animais	20	36000
Serengeti - The Adventure	Animais	30	45000
Force Volkswagen Commercial	Carro	1	1500
Ferrari 458 Spider	Carro	3	4500
Exotic Car Traffic at Gas Station	Carro	5	7500
Best of Supercar Sounds	Carro	10	15000
All of the Cars in NFS Most Wanted	Carro	20	36000
Grid 2 - The First 30 Minutes Gameplay Footage	Carro	30	54000
Parabéns	Comédia	1	1439
Despedida de Solteiro	Comédia	3	4316
Funny Videos Compilation	Comédia	5	8992
DeadPool Funny Highlights	Comédia	10	17983
Portaria 31 - Pessoa Ruim	Comédia	20	28772
Entrevista Comédia	Comédia	30	43154
Galinha Pintadinha Faixa 1	Desenho	1	1799
The Colors of Evil	Desenho	3	4320
Aventura na Floresta	Desenho	5	9000
Sítio do Picapau Amarelo	Desenho	10	17983
Magic Kingdom	Desenho	20	35965
Animated Short Film / Movie Compilation	Desenho	30	43200
Programa do Jô	Entrevista	1	1799
De Frente com Gabi - Entrevista com Alexandre Pato	Entrevista	3	5395
De Frente com Gabi - Oscar Schmidt - Parte 2	Entrevista	5	8992
De Frente com Gabi - Oscar Schmidt - Parte 1	Entrevista	10	18000
Clint Eastwood, Morgan Freeman and Hilary Swank	Entrevista	20	36000
Arucitys - Entrevista a Nuria Roca Y Juan del Val	Entrevista	30	53947
Sports Action Camera Test	Esporte	1	1799
Red Bull Sport	Esporte	3	5400
Extreme Sports	Esporte	5	9000
UFC 3 Matt Hughes vs Royce Gracie Gameplay	Esporte	10	17983
Cristiano Ronaldo - A Golden Year 2013	Esporte	20	30000
Brazilian Fighters	Esporte	30	53947
Era do Gelo 4 Dublado	Filme	1	1439
Elysium Trailer #2 2013 Official - Matt Damon Movie	Filme	3	4316
Ultimate Matrix trilogy 6 min trailer	Filme	5	7500
Tears of Steel Cyberpunk Action Scifi 3D Animation	Filme	10	14400
P. S. I Love You - Parts	Filme	20	30000
StarCraft 2 Heart of the Swarm All Cinematics	Filme	30	53947
Cândida Oliveira - Band News	Jornal	1	1799
Susanna Reid BBC breakfast 23rd Jan 2013	Jornal	3	4500
BBC News	Jornal	5	8992
Ron Paul Speech CNN Focus Group Reactions	Jornal	10	17983
Ron Raul South Carolina CNN Debate 1/19/2012	Jornal	20	35965
ABC News 24 - 1:00pm Bulletin Highlights	Jornal	30	45000
High School Musical 3 - Can I have This Dance	Música	1	1500
Linkin Park - What I've Done	Música	3	5400
SNSD - The Boys	Música	5	8992
The Cranberries - Tomorrow, Zombie	Música	10	17983
Taylor Swift	Música	20	35965
Taylor Swift	Música	30	53947
Heavenly Mountain Sunrise Timelapse 1 minute	Natureza	1	1799
Amazing nature	Natureza	3	4500
Nature	Natureza	5	7500
New Zealand	Natureza	10	14386
Flowers	Natureza	20	36000
McWay Falls, Big Sur	Natureza	30	45000

ximo a zero. Como esperado, o tempo médio de execução cresce proporcionalmente à duração e à resolução do vídeo.

Tabela 5.2: Resultados VSUMM - tempo de execução e desvio padrão versão sequencial

Resultados VSUMM - Versão Sequencial					
Resolução	**Duração (min)**	**Segmentação do Vídeo**	**Extração de Características**	**Agrupamento dos Quadros**	**Total**
320 × 240	1	0.655(0.33)	0.223(0.01)	0.002(0.00)	0.980(0.36)
320 × 240	3	1.717(0.38)	0.650(0.02)	0.017(0.01)	2.540(0.41)
320 × 240	5	2.609(0.48)	1.123(0.05)	0.046(0.03)	3.978(0.49)
320 × 240	10	4.701(0.47)	2.253(0.09)	0.159(0.08)	7.489(0.54)
320 × 240	20	9.777(0.84)	5.215(1.99)	0.429(0.41)	15.796(2.07)
320 × 240	30	14.173(0.94)	6.769(0.32)	1.099(1.49)	22.673(1.88)
640 × 360	1	0.892(0.33)	0.506(0.02)	0.002(0.00)	1.493(0.36)
640 × 360	3	2.375(0.31)	1.480(0.05)	0.018(0.01)	4.025(0.37)
640 × 360	5	3.761(0.39)	2.520(0.12)	0.034(0.03)	6.533(0.50)
640 × 360	10	7.127(0.60)	5.124(0.20)	0.134(0.06)	12.720(0.76)
640 × 360	20	15.528(1.66)	10.501(0.70)	0.442(0.38)	26.872(2.17)
640 × 360	30	21.693(1.34)	15.301(0.85)	0.828(0.52)	38.346(2.00)
1280 × 720	1	2.153(0.41)	1.896(0.12)	0.002(0.00)	4.160(0.50)
1280 × 720	3	6.022(0.54)	5.563(0.16)	0.014(0.01)	11.811(0.67)
1280 × 720	5	9.879(0.81)	9.557(0.47)	0.040(0.03)	19.673(1.24)
1280 × 720	10	19.816(1.71)	19.376(0.75)	0.185(0.20)	39.644(2.43)
1280 × 720	20	42.190(5.35)	39.747(2.83)	0.555(0.62)	82.861(8.12)
1280 × 720	30	59.931(4.45)	57.233(1.70)	0.937(0.71)	118.680(5.97)
1920 × 1080	1	4.085(0.53)	4.149(0.23)	0.002(0.00)	8.379(0.79)
1920 × 1080	3	11.618(0.72)	12.207(0.41)	0.013(0.01)	23.992(0.88)
1920 × 1080	5	19.629(1.69)	20.962(0.95)	0.055(0.04)	40.838(2.59)
1920 × 1080	10	39.624(3.43)	42.443(1.57)	0.124(0.08)	82.480(5.05)
1920 × 1080	20	83.573(11.51)	87.363(5.83)	0.352(0.29)	171.669(17.21)
1920 × 1080	30	117.580(7.13)	125.634(4.43)	1.115(0.66)	244.932(11.05)

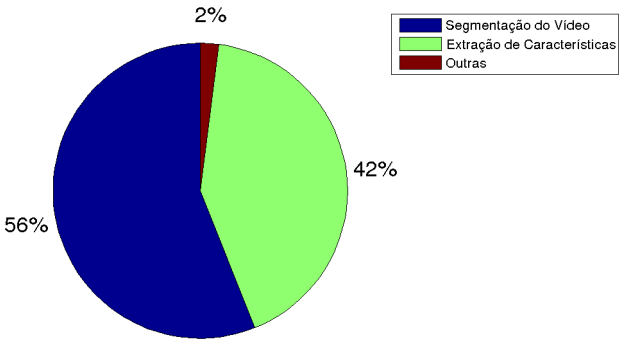


Figura 5.1: Porcentagem do tempo gasto nas etapas do VSUMM na versão sequencial

O gráfico da Figura 5.1 demonstra a porcentagem de tempo das etapas em relação ao tempo total de execução. Claramente é possível concluir que a etapa de

segmentação do vídeo é a mais custosa (56%), seguida pela extração de características (42%). As outras três etapas contribuem com uma porcentagem mínima de tempo de execução, próxima a 2%.

5.2.2 Experimentos Versões, *Multicore* CPU e GPUs

Inicialmente a versão *multicore* CPU foi avaliada através da comparação entre as bibliotecas OpenMP e *threads C++11*, em busca da melhor para o tipo de algoritmo em análise, pois ambas possuem otimizações próprias e específicas em suas implementações. As tabelas 5.3 e 5.4 apresentam os tempos médio de execução e desvio padrão das implementações utilizando OpenMP e *threads C++11*, respectivamente. Novamente, apenas o tempo das três primeiras etapas do VSUMM e o total são apresentadas nas tabelas, ocultando o tempo das duas últimas etapas por ser próximo a zero.

Tabela 5.3: Resultados VSUMM - tempo de execução e desvio padrão versão *multicore* CPU OpenMP

Resultados VSUMM - Versão Multicore CPU - OMP					
Resolução	**Duração (min)**	**Segmentação do Vídeo**	**Extração de Características**	**Agrupamento dos Quadros**	**Total**
320 × 240	1	0.548(0.59)	0.056(0.01)	0.001(0.00)	0.682(0.60)
320 × 240	3	1.212(0.63)	0.159(0.02)	0.005(0.00)	1.563(0.64)
320 × 240	5	1.347(0.64)	0.253(0.03)	0.013(0.01)	1.842(0.62)
320 × 240	10	1.815(0.61)	0.484(0.03)	0.055(0.03)	2.642(0.58)
320 × 240	20	3.507(0.69)	0.942(0.06)	0.204(0.26)	5.073(0.63)
320 × 240	30	4.755(0.59)	1.415(0.06)	0.385(0.29)	7.872(0.92)
640 × 360	1	0.692(0.58)	0.152(0.03)	0.001(0.00)	0.937(0.58)
640 × 360	3	1.556(0.65)	0.418(0.03)	0.005(0.00)	2.159(0.65)
640 × 360	5	1.899(0.65)	0.672(0.05)	0.013(0.01)	2.815(0.66)
640 × 360	10	2.970(0.65)	1.357(0.06)	0.056(0.03)	4.692(0.66)
640 × 360	20	5.983(0.73)	2.879(0.27)	0.240(0.28)	12.200(1.88)
640 × 360	30	8.541(0.84)	4.242(0.23)	0.376(0.26)	16.763(2.37)
1280 × 720	1	1.415(0.60)	0.533(0.02)	0.001(0.00)	2.030(0.62)
1280 × 720	3	3.411(0.66)	1.615(0.06)	0.006(0.00)	5.221(0.70)
1280 × 720	5	4.921(0.74)	2.737(0.10)	0.017(0.02)	8.309(0.84)
1280 × 720	10	9.009(0.86)	5.715(0.26)	0.043(0.02)	15.664(0.83)
1280 × 720	20	20.146(2.78)	11.276(0.47)	0.201(0.23)	32.241(3.09)
1280 × 720	30	33.180(3.42)	16.378(0.51)	0.362(0.28)	50.579(3.61)
1920 × 1080	1	2.633(0.59)	1.173(0.04)	0.001(0.00)	3.905(0.60)
1920 × 1080	3	6.570(0.67)	3.547(0.12)	0.004(0.00)	10.500(0.68)
1920 × 1080	5	10.142(1.14)	6.055(0.27)	0.013(0.01)	16.628(1.36)
1920 × 1080	10	19.351(1.30)	12.226(0.40)	0.041(0.03)	32.116(1.41)
1920 × 1080	20	40.466(4.22)	24.736(1.15)	0.147(0.14)	65.846(5.10)
1920 × 1080	30	57.492(3.07)	36.522(0.95)	0.448(0.29)	95.311(3.70)

Os tempos de execução apresentados nas tabelas 5.3 e 5.4, possuem uma variação mínima entre si, sendo na maioria dos casos diferenças nas casas decimais. Com isso, não é possível concluir a melhor biblioteca entre as analisadas, ambas apre-

Tabela 5.4: Resultados VSUMM - tempo de execução e desvio padrão versão *multicore* CPU *Threads C++11*

Resultados VSUMM - Versão Multicore CPU - Threads					
Resolução	**Duração (min)**	**Segmentação do Vídeo**	**Extração de Características**	**Agrupamento dos Quadros**	**Total**
320 × 240	1	0.534(0.58)	0.075(0.01)	0.002(0.00)	0.706(0.61)
320 × 240	3	1.160(0.60)	0.187(0.02)	0.008(0.00)	1.536(0.62)
320 × 240	5	1.299(0.63)	0.292(0.02)	0.013(0.00)	1.833(0.67)
320 × 240	10	1.762(0.71)	0.545(0.04)	0.052(0.03)	2.648(0.71)
320 × 240	20	3.502(0.67)	1.038(0.09)	0.129(0.12)	5.050(0.62)
320 × 240	30	4.721(0.64)	1.594(0.21)	0.279(0.22)	8.002(2.75)
640 × 360	1	0.683(0.59)	0.187(0.02)	0.002(0.00)	0.962(0.59)
640 × 360	3	1.533(0.61)	0.477(0.03)	0.009(0.00)	2.202(0.60)
640 × 360	5	1.914(0.62)	0.749(0.04)	0.012(0.01)	2.897(0.64)
640 × 360	10	2.937(0.64)	1.401(0.07)	0.043(0.02)	4.657(0.68)
640 × 360	20	5.976(0.85)	2.991(0.27)	0.158(0.16)	11.456(2.78)
640 × 360	30	8.235(0.54)	4.359(0.26)	0.272(0.16)	16.710(1.19)
1280 × 720	1	1.425(0.60)	0.689(0.04)	0.003(0.00)	2.204(0.61)
1280 × 720	3	3.379(0.62)	1.756(0.06)	0.008(0.00)	5.377(0.70)
1280 × 720	5	4.906(0.73)	2.919(0.11)	0.018(0.01)	8.352(1.04)
1280 × 720	10	8.776(0.80)	5.973(0.41)	0.044(0.02)	15.551(1.00)
1280 × 720	20	20.499(3.37)	11.527(0.66)	0.156(0.16)	33.120(3.81)
1280 × 720	30	28.713(2.42)	16.895(0.55)	0.266(0.19)	46.997(2.75)
1920 × 1080	1	2.663(0.62)	1.501(0.08)	0.002(0.00)	4.254(0.59)
1920 × 1080	3	6.577(0.68)	3.829(0.13)	0.007(0.00)	11.095(0.77)
1920 × 1080	5	10.089(1.01)	6.394(0.28)	0.017(0.01)	17.125(1.19)
1920 × 1080	10	18.925(1.36)	12.284(0.43)	0.041(0.02)	32.075(1.68)
1920 × 1080	20	40.335(4.10)	25.061(1.41)	0.132(0.11)	66.204(5.06)
1920 × 1080	30	59.098(3.22)	36.839(1.25)	0.363(0.21)	97.191(3.61)

sentaram resultados satisfatórios e aproximados. Para os próximos experimentos foi escolhida a implementação que utiliza *threads C++11*, por nenhuma razão específica. Observe que o maior ganho das versões *multicore* CPU em relação à versão sequencial foi na etapa de segmentação do vídeo. Ao final deste capítulo é apresenta uma análise mais detalhada em relação à melhoria do tempo de execução de todas as versões.

Para avaliar qual a melhor implementação em GPU, entre as duas formas apresentadas, o tempo de execução também foi analisado. As tabelas 5.5 e 5.6 mostram os resultados desses experimentos. A diferença entre as duas implementações é o método de segmentação do vídeo. Na primeira, a segmentação ocorre totalmente em CPU, enquanto a segunda consiste em decodificar os quadros em GPU e copiá-los para CPU. A comparação entre essas baseia-se na análise do impacto que a segmentação do vídeo tem para o método e em como a cópia de GPU para CPU influencia nos resultados. Os resultados então mostram que a cópia da GPU para CPU realmente influencia no tempo de execução do algoritmo, deixando não só a etapa de decodificação pior do que em CPU, como o tempo total do método. Já a etapa de extração de características, igual nessas duas implementações, apresentou uma

redução significativa do tempo de execução em relação à versão sequencial,

Tabela 5.5: Resultados VSUMM - tempo de execução e desvio padrão versão GPU tipo 1

Resultados VSUMM - Versão GPU - Tipo 1					
Resolução	**Duração (min)**	**Segmentação do Vídeo**	**Extração de Características**	**Agrupamento dos Quadros**	**Total**
320 × 240	1	0.727(0.32)	0.017(0.00)	0.003(0.00)	0.853(0.34)
320 × 240	3	1.797(0.36)	0.045(0.00)	0.003(0.00)	2.010(0.38)
320 × 240	5	2.735(0.35)	0.074(0.00)	0.004(0.00)	3.014(0.36)
320 × 240	10	4.919(0.34)	0.145(0.00)	0.006(0.00)	5.517(0.36)
320 × 240	20	10.542(1.40)	0.286(0.00)	0.006(0.00)	11.613(1.52)
320 × 240	30	20.483(3.83)	0.430(0.01)	0.008(0.00)	21.986(3.58)
640 × 360	1	0.981(0.32)	0.045(0.00)	0.003(0.00)	1.133(0.34)
640 × 360	3	2.471(0.40)	0.132(0.00)	0.004(0.00)	2.790(0.42)
640 × 360	5	3.902(0.49)	0.218(0.00)	0.004(0.00)	4.406(0.60)
640 × 360	10	7.703(0.89)	0.432(0.00)	0.006(0.00)	8.854(1.09)
640 × 360	20	22.983(5.19)	0.863(0.01)	0.006(0.00)	24.997(4.92)
640 × 360	30	31.956(5.37)	1.305(0.02)	0.007(0.00)	38.262(5.93)
1280 × 720	1	2.243(0.42)	0.173(0.00)	0.003(0.00)	2.561(0.45)
1280 × 720	3	6.449(0.66)	0.517(0.01)	0.004(0.00)	7.371(0.61)
1280 × 720	5	10.403(1.13)	0.857(0.01)	0.004(0.00)	11.668(1.11)
1280 × 720	10	22.315(2.63)	1.783(0.22)	0.005(0.00)	28.702(2.87)
1280 × 720	20	48.874(7.77)	3.403(0.05)	0.006(0.00)	65.157(9.05)
1280 × 720	30	73.073(14.48)	5.157(0.07)	0.007(0.00)	98.316(14.48)
1920 × 1080	1	4.144(0.51)	0.387(0.01)	0.003(0.00)	4.720(0.56)
1920 × 1080	3	12.120(1.00)	1.165(0.01)	0.004(0.00)	17.296(1.19)
1920 × 1080	5	21.299(2.64)	1.935(0.03)	0.004(0.00)	31.009(3.05)
1920 × 1080	10	42.995(4.23)	3.854(0.04)	0.005(0.00)	62.360(5.26)
1920 × 1080	20	89.762(10.59)	7.653(0.11)	0.006(0.00)	129.505(13.90)
1920 × 1080	30	127.462(8.98)	11.594(0.18)	0.007(0.00)	188.240(11.80)

A partir desses resultados, foram executados experimentos comparando a versão sequencial com a melhor implementação em *multicore* CPU, utilizando *threads C++11*, e em GPU, com a segmentação em CPU e extração de características em GPU. Os gráficos das Figura 5.2 apresentam os resultados em relação ao *speedup* obtido com as duas principais e mais custosas etapas do VSUMM (segmentação do vídeo e extração de características).

Em relação à segmentação do vídeo, a versão GPU não apresentou *speedup* em relação à versão sequencial pois a mesma implementação foi utilizada nas duas versões. Já a versão *multicore* apresentou melhorias de aproximadamente 3× para todas as resoluções. A extração de características em GPU e *multicore* CPU apresentaram ganho de tempo de execução em relação à versão sequencial. Nesse caso, a versão GPU apresentou resultados até 18× melhores do que a versão sequencial e a versão *multicore* obteve *speedup* de até 5×.

Tabela 5.6: Resultados VSUMM - tempo de execução e desvio padrão versão GPU tipo 2

Resultados VSUMM - Versão GPU - Tipo 2					
Resolução	**Duração (min)**	**Segmentação do Vídeo**	**Extração de Características**	**Agrupamento dos Quadros**	**Total**
320 × 240	1	0.727(0.32)	0.017(0.00)	0.003(0.00)	0.853(0.34)
320 × 240	3	1.797(0.36)	0.045(0.00)	0.003(0.00)	2.010(0.38)
320 × 240	5	2.735(0.35)	0.074(0.00)	0.004(0.00)	3.014(0.36)
320 × 240	10	4.919(0.34)	0.145(0.00)	0.006(0.00)	5.517(0.36)
320 × 240	20	10.542(1.40)	0.286(0.00)	0.006(0.00)	11.613(1.52)
320 × 240	30	20.483(3.83)	0.430(0.01)	0.008(0.00)	21.986(3.58)
640 × 360	1	0.981(0.32)	0.045(0.00)	0.003(0.00)	1.133(0.34)
640 × 360	3	2.471(0.40)	0.132(0.00)	0.004(0.00)	2.790(0.42)
640 × 360	5	3.902(0.49)	0.218(0.00)	0.004(0.00)	4.406(0.60)
640 × 360	10	7.703(0.89)	0.432(0.00)	0.006(0.00)	8.854(1.09)
640 × 360	20	22.983(5.19)	0.863(0.01)	0.006(0.00)	24.997(4.92)
640 × 360	30	31.956(5.37)	1.305(0.02)	0.007(0.00)	38.262(5.93)
1280 × 720	1	2.243(0.42)	0.173(0.00)	0.003(0.00)	2.561(0.45)
1280 × 720	3	6.449(0.66)	0.517(0.01)	0.004(0.00)	7.371(0.61)
1280 × 720	5	10.403(1.13)	0.857(0.01)	0.004(0.00)	11.668(1.11)
1280 × 720	10	22.315(2.63)	1.783(0.22)	0.005(0.00)	28.702(2.87)
1280 × 720	20	48.874(7.77)	3.403(0.05)	0.006(0.00)	65.157(9.05)
1280 × 720	30	73.073(14.48)	5.157(0.07)	0.007(0.00)	98.316(14.48)
1920 × 1080	1	4.144(0.51)	0.387(0.01)	0.003(0.00)	4.720(0.56)
1920 × 1080	3	12.120(1.00)	1.165(0.01)	0.004(0.00)	17.296(1.19)
1920 × 1080	5	21.299(2.64)	1.935(0.03)	0.004(0.00)	31.009(3.05)
1920 × 1080	10	42.995(4.23)	3.854(0.04)	0.005(0.00)	62.360(5.26)
1920 × 1080	20	89.762(10.59)	7.653(0.11)	0.006(0.00)	129.505(13.90)
1920 × 1080	30	127.462(8.98)	11.594(0.18)	0.007(0.00)	188.240(11.80)
1920 × 1080	30	322.905(33.70)	11.763(0.27)	0.010(0.01)	386.653(33.55)

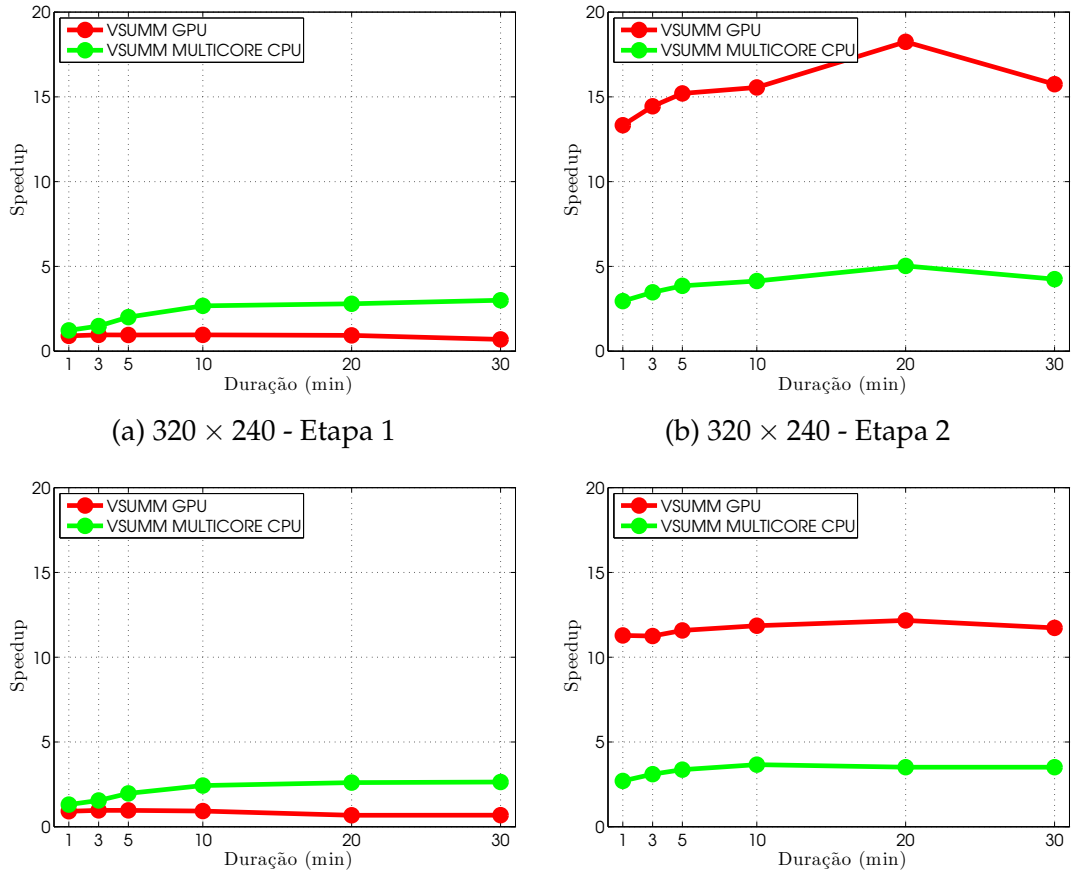
5.2.3 Implementação e Experimentos para a Versão Híbrida

Ao comparar os resultados obtidos para cada etapa do VSUMM nos experimentos com as versões *multicore* CPU e GPU, é possível criar uma versão híbrida, composta por cada etapa da versão que apresentou os melhores tempos de execução. Esta nova versão é apresentada a seguir, juntamente com os experimentos relacionados a ela.

5.2.3.1 Implementação Versão Híbrida

A seguir são apresentadas as versões escolhidas para cada etapa da versão híbrida, juntamente com a explicação dessas escolhas, também apresentadas na Figura 5.3.

- Segmentação/Decodificação do Vídeo: *multicore* CPU. Como discutido nos experimentos, a decodificação do vídeo em quadros utilizando várias *threads* da versão *multicore* CPU apresentou resultados melhores do que as versões desta etapa implementadas em GPU.



(a) 320 × 240 - Etapa 1

(b) 320 × 240 - Etapa 2

(c) 640 × 360 - Etapa 1

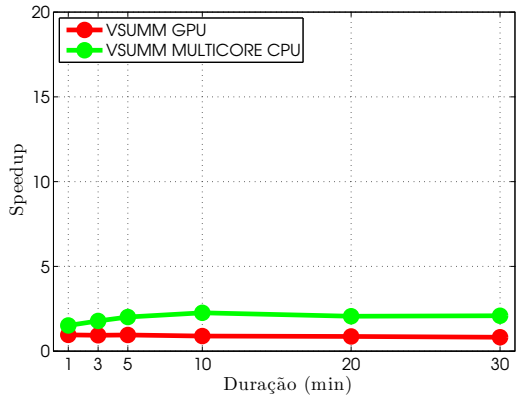
(d) 640 × 360 - Etapa 2

Figura 5.2: *Speedups* obtidos pelas versões *multicore* CPU e GPU nas etapas 1 (segmentação do vídeo) e 2 (extração de características) com as quatros resoluções analisadas.

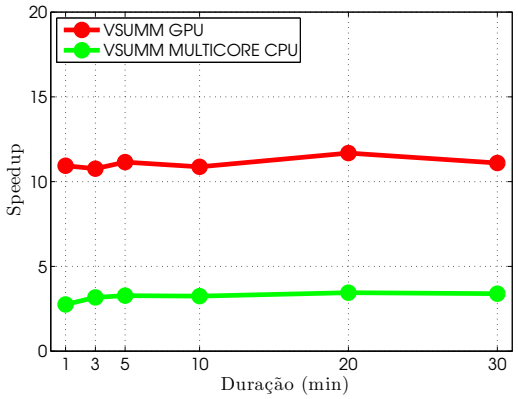
- Extração de Características e Eliminação de Quadros Inexpressivos: GPU. Como esta etapa possui muitos cálculos matemáticos repetitivos e simples, as GPUs são ideais e mais eficientes, explicando os melhores resultados para esta etapa.

- Agrupamento dos Quadros: GPU. Todas as versões apresentaram resultados rápidos nessa etapa, porém a versão GPU atingiu tempos de execução ligeiramente mais rápidos que as outras versões.

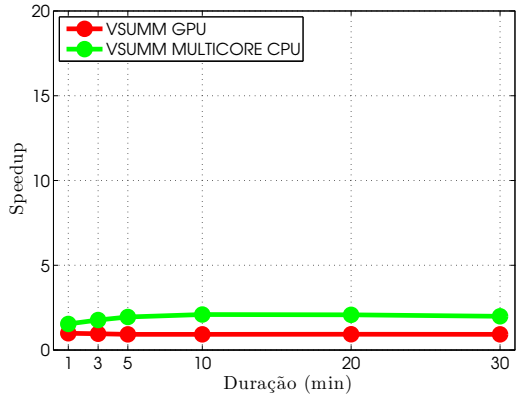
Dada essa versão híbrida, foram realizados testes para comprovar que ela é melhor que as duas outras versões implementadas. Nesses testes, foi medida a quantidade mínima, máxima e média de quadros processados por segundo (FPS) em todas as implementações. Além disso, essa métrica foi analisada por gênero, duração e resolução, por exemplo, para um mesmo gênero, qual a quantidade mí-



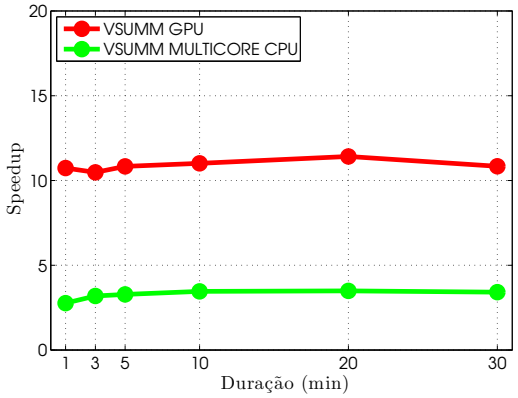
(e) 1280 × 720 - Etapa 1



(f) 1280 × 720 - Etapa 2



(g) 1920 × 1080 - Etapa 1



(h) 1920 × 1080 - Etapa 2

Figura 5.2: *Speedups* obtidos pelas versões *multicore* CPU e GPU nas etapas 1 (segmentação do vídeo) e 2 (extração de características) com as quatros resoluções analisadas.

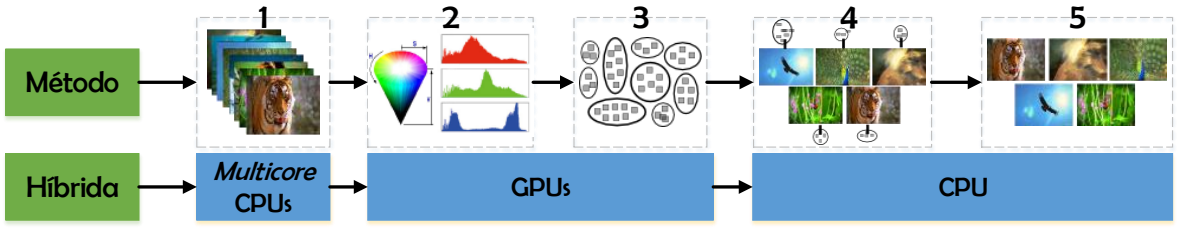


Figura 5.3: Diagrama de Fluxo de Dados/Execução da Implementação Híbrida para a Abordagem de Sumarização de Vídeos proposta em Avila et al. [2011]

nima, máxima e média de quadros processados por segundo. As tabelas 5.7, 5.8 e 5.9 mostram os resultados desses experimentos.

Em relação aos gêneros dos vídeos, a Tabela 5.7 mostra que os resultados são semelhantes para os diferentes tipos de vídeos, principalmente na quantidade mí-

Tabela 5.7: Efetividade em quadros processados por segundo para as versões implementadas em relação aos gêneros dos vídeos.

Resultados VSUMM - Quadros Processados por Segundo por Gênero				
[redacted]	**Gênero**	**FPS Min**	**FPS Max**	**FPS Médio**
CPU	Animais	6.00	82.00	34.00
CPU	Carro	6.00	82.00	31.92
CPU	Desenho	6.00	80.00	33.96
CPU	Comédia	7.00	88.00	39.33
CPU	Entrevista	7.00	90.00	38.71
CPU	Filme	7.00	87.00	37.25
CPU	Música	6.00	84.00	34.00
CPU	Natureza	7.00	86.00	37.12
CPU	Jornal	7.00	84.00	36.83
CPU	Esporte	7.00	84.00	35.50
GPU	Animais	9.00	113.00	44.75
GPU	Carro	8.00	98.00	40.33
GPU	Desenho	8.00	119.00	45.71
GPU	Comédia	10.00	117.00	52.62
GPU	Entrevista	10.00	127.00	52.33
GPU	Filme	9.00	112.00	50.67
GPU	Música	9.00	122.00	45.33
GPU	Natureza	9.00	111.00	47.25
GPU	Jornal	9.00	110.00	45.79
GPU	Esporte	8.00	111.00	43.29
Multicore CPU	Animais	10.00	222.00	67.38
Multicore CPU	Carro	9.00	171.00	52.79
Multicore CPU	Desenho	8.00	216.00	62.71
Multicore CPU	Comédia	12.00	184.00	68.62
Multicore CPU	Entrevista	11.00	178.00	61.58
Multicore CPU	Filme	12.00	183.00	65.92
Multicore CPU	Música	10.00	220.00	65.50
Multicore CPU	Natureza	11.00	208.00	65.62
Multicore CPU	Jornal	12.00	219.00	73.79
Multicore CPU	Esporte	10.00	210.00	57.79
Híbrida	Animais	12.00	272.00	87.67
Híbrida	Carro	12.00	235.00	68.38
Híbrida	Desenho	15.00	265.00	85.00
Híbrida	Comédia	15.00	276.00	104.29
Híbrida	Entrevista	16.00	284.00	97.96
Híbrida	Filme	15.00	253.00	86.38
Híbrida	Música	11.00	275.00	85.71
Híbrida	Natureza	16.00	263.00	86.21
Híbrida	Jornal	16.00	295.00	102.00
Híbrida	Esporte	15.00	287.00	84.71

nima e máxima de quadros processados por segundo. No entanto, o FPS médio apresentou alguma disparidades, em especial na versão híbrida. Os vídeos do gênero de carro atingiram o menor FPS em todas as versões, enquanto os vídeos de comédia obtiveram os maiores FPS. Como esperado, a versão híbrida atingiu os maiores FPS em todos os gêneros, chegando a um máximo de 295 quadros por segundo.

Analisando a quantidade de quadros processados por segundo com relação à duração dos vídeos, a Tabela 5.8 também apresenta resultados semelhantes no FPS mínimo e máximo, e algumas variações no FPS médio. Esses resultados mostram

Tabela 5.8: Efetividade em quadros processados por segundo para as versões implementadas em relação às durações dos vídeos.

Resultados VSUMM - Quadros Processados por Segundo por Duração				
Versão	**Duração**	**FPS Min**	**FPS Max**	**FPS Médio**
CPU	1	6.00	86.00	32.90
CPU	3	7.00	90.00	35.20
CPU	5	6.00	87.00	36.30
CPU	10	7.00	86.00	37.55
CPU	20	6.00	84.00	35.88
CPU	30	7.00	88.00	37.35
GPU	1	10.00	106.00	43.12
GPU	3	9.00	127.00	48.55
GPU	5	8.00	123.00	51.40
GPU	10	8.00	119.00	52.15
GPU	20	8.00	122.00	45.65
GPU	30	9.00	110.00	39.98
Multicore CPU	1	10.00	163.00	51.67
Multicore CPU	3	12.00	178.00	52.80
Multicore CPU	5	12.00	208.00	69.85
Multicore CPU	10	10.00	222.00	82.40
Multicore CPU	20	8.00	220.00	65.62
Multicore CPU	30	9.00	189.00	62.67
Híbrida	1	11.00	232.00	66.38
Híbrida	3	15.00	238.00	70.20
Híbrida	5	15.00	261.00	86.95
Híbrida	10	17.00	287.00	106.88
Híbrida	20	16.00	295.00	101.38
Híbrida	30	17.00	284.00	101.20

que a duração do vídeo não influencia da quantidade de quadros processados por segundo, apenas no tempo total de execução do método.

Tabela 5.9: Efetividade em quadros processados por segundo para as versões implementadas em relação às resoluções dos vídeos.

Resultados VSUMM - Quadros Processados por Segundo por Resolução				
Versão	**Resolução**	**FPS Min**	**FPS Max**	**FPS Médio**
CPU	320×240	39.00	90.00	75.70
CPU	640×360	29.00	53.00	45.48
CPU	1280×720	12.00	18.00	15.00
CPU	1920×1080	6.00	9.00	7.27
GPU	320×240	42.00	127.00	95.12
GPU	640×360	35.00	84.00	59.72
GPU	1280×720	13.00	30.00	22.18
GPU	1920×1080	8.00	15.00	10.22
Multicore CPU	320×240	35.00	222.00	141.28
Multicore CPU	640×360	30.00	116.00	76.77
Multicore CPU	1280×720	17.00	34.00	26.15
Multicore CPU	1920×1080	8.00	15.00	12.48
Híbrida	320×240	36.00	295.00	198.22
Híbrida	640×360	32.00	151.00	104.30
Híbrida	1280×720	19.00	43.00	35.48
Híbrida	1920×1080	11.00	20.00	17.32

Os resultados apresentados na Tabela 5.9 mostram que este é o fator que mais influencia no FPS mínimo, máximo e médio. Ou seja, enquanto o FPS por gênero e duração não sofreu grandes alterações entre si, o FPS muda muito de uma resolução para outra. As menores resoluções tem maior FPS pois os quadros são mais fáceis de processar e as maiores resoluções apresentam menor FPS, tanto mínimo, máximo quanto médio.

5.3 Experimentos para a Abordagem TEMPSUMM [Cahuina et al., 2013]

Novamente, foram realizados experimentos para analisar a eficiência das versões *multicore* CPU e GPU e então gerar uma versão híbrida. Esses experimentos são apresentados a seguir.

5.3.1 Experimentos Versão Sequencial

Inicialmente foram realizados experimentos medindo o tempo de execução e desvio padrão da versão sequencial. A Tabela 5.10 apresenta os resultados para as cinco primeiras etapas do TEMPSUMM, e o tempo total de execução. As duas últimas etapas do método foram ocultadas da tabela pois são foram paralelizadas, então não serão utilizadas na comparação com as versões paralelas. É importante observar que as etapas 3 e 4 (detecção de quadros representativos e *bag-of-words*) apresentam maior desvio padrão em relação às outras etapas. Este fato ocorre pois essas etapas utilizam algoritmos de agrupamento, *x-means* e LBG, respectivamente, e esse tipo de algoritmo pode variar muito de um conjunto de dados para outro em relação à rapidez da formação dos agrupamentos.

O gráfico da Figura 5.4 apresenta as porcentagens em relação ao tempo total de execução obtidas pelas etapas do método de sumarização. Claramente a etapa 4 representa o maior tempo de execução entre as etapas do algoritmo. As outras etapas contribuem com parcelas próximas do tempo total, entre 11% e 15%. A etapa menos custosa é a de filtragem, responsável por menos de 1% do tempo total de execução.

5.3.2 Experimentos Versões *Multicore* CPU e GPUs

Para testar a versão *multicore* CPU, o método de sumarização foi implementado utilizando duas bibliotecas para processamento *multithread*, OpenMP e *threads C++11*.

Tabela 5.10: Resultados TEMPSUMM - tempo de execução e desvio padrão versão sequencial

Resultados TEMPSUMM - Versão Sequencial							
Resolução	**Duração (min)**	**Etapa 1**	**Etapa 2**	**Etapa 3**	**Etapa 4**	**Etapa 5**	**Total**
320 × 240	1	0.762(0.45)	0.145(0.01)	5.020(1.40)	50.307(146.43)	0.273(0.07)	57.295(146.47)
320 × 240	3	1.825(0.32)	0.418(0.03)	17.147(6.04)	2.622(0.86)	0.901(0.33)	26.997(7.35)
320 × 240	5	2.652(0.34)	0.715(0.04)	14.783(9.07)	5.103(3.33)	0.768(0.48)	26.285(12.27)
320 × 240	10	5.004(0.31)	1.433(0.11)	203.828(539.45)	5.059(3.02)	1.858(1.08)	240.001(537.75)
320 × 240	20	10.436(0.44)	3.076(0.26)	130.618(205.06)	4.042(0.08)	3.279(2.25)	160.935(207.83)
320 × 240	30	16.253(1.18)	4.474(0.39)	2709.820(6739.25)	13.875(8.38)	4.792(2.87)	2753.920(6744.98)
640 × 360	1	0.904(0.33)	0.356(0.02)	5.271(1.45)	3.989(0.07)	0.831(0.18)	13.798(2.08)
640 × 360	3	2.476(0.36)	1.034(0.06)	17.680(6.98)	8.418(2.98)	2.791(1.06)	36.589(11.20)
640 × 360	5	3.918(0.31)	1.774(0.10)	15.791(7.58)	486.120(1521.05)	2.371(1.25)	516.993(1526.74)
640 × 360	10	7.602(0.54)	3.842(0.57)	33.875(17.07)	14.957(8.07)	5.178(2.77)	69.967(27.72)
640 × 360	20	18.957(3.76)	7.594(0.59)	382.789(1013.43)	4.078(0.29)	9.619(7.17)	450.781(1017.21)
640 × 360	30	29.291(5.00)	10.920(0.56)	105.491(73.49)	41.151(27.17)	14.641(9.61)	871.835(2053.77)
1280 × 720	1	2.227(0.37)	1.358(0.09)	6.838(2.00)	6.009(6.32)	3.343(1.14)	30.459(9.27)
1280 × 720	3	6.379(0.62)	4.059(0.25)	23.239(9.35)	35.486(12.07)	11.193(3.88)	85.577(25.00)
1280 × 720	5	10.718(0.97)	6.768(0.31)	20.873(9.34)	4.006(0.10)	9.549(4.11)	83.293(25.75)
1280 × 720	10	21.360(2.57)	13.595(0.93)	55.979(48.47)	67.450(34.67)	21.594(11.58)	185.759(89.95)
1280 × 720	20	46.456(6.87)	28.386(2.34)	90.909(59.91)	19.835(49.95)	40.409(28.10)	354.029(194.21)
1280 × 720	30	70.093(13.36)	40.133(1.63)	130.419(75.47)	200.045(121.99)	61.175(37.47)	511.220(236.56)
1920 × 1080	1	4.248(0.52)	2.933(0.14)	9.004(1.98)	79.554(238.97)	6.932(1.25)	125.961(237.96)
1920 × 1080	3	12.164(0.78)	8.606(0.36)	31.383(12.06)	74.459(28.03)	24.854(8.99)	157.066(49.35)
1920 × 1080	5	20.348(1.13)	14.571(0.62)	28.631(15.40)	214.640(666.11)	20.393(10.12)	359.919(656.83)
1920 × 1080	10	41.961(3.63)	29.441(1.81)	60.093(32.03)	143.006(69.93)	47.163(25.77)	329.248(127.82)
1920 × 1080	20	88.497(9.56)	62.126(5.15)	111.378(73.62)	4.227(0.59)	85.926(56.39)	582.568(287.74)
1920 × 1080	30	126.534(10.38)	87.696(3.57)	186.589(123.31)	376.979(243.23)	134.396(85.49)	927.269(450.65)

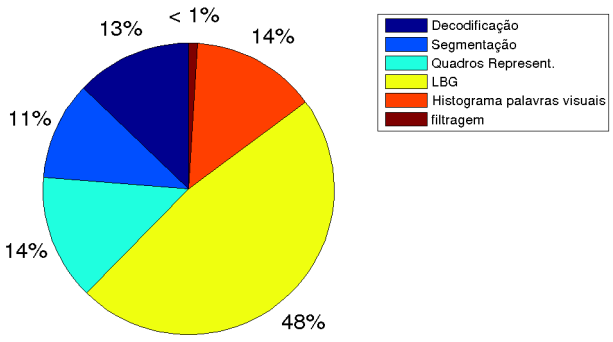


Figura 5.4: Porcentagem do tempo gasto nas etapas do TEMPSUMM na versão sequencial

Como no VSUMM, os resultados foram similares nas duas implementações. Devido a este fato, serão apresentados nesta subseção apenas os resultados obtidos com o biblioteca *threads C++11*.

A Tabela 5.11 contém esses resultados. É possível concluir a partir desses dados que todas as etapas foram aceleradas na versão em análise. No entanto, as etapas que já apresentavam um alto desvio padrão continuaram apresentando, influenciando muito no desvio padrão do tempo total.

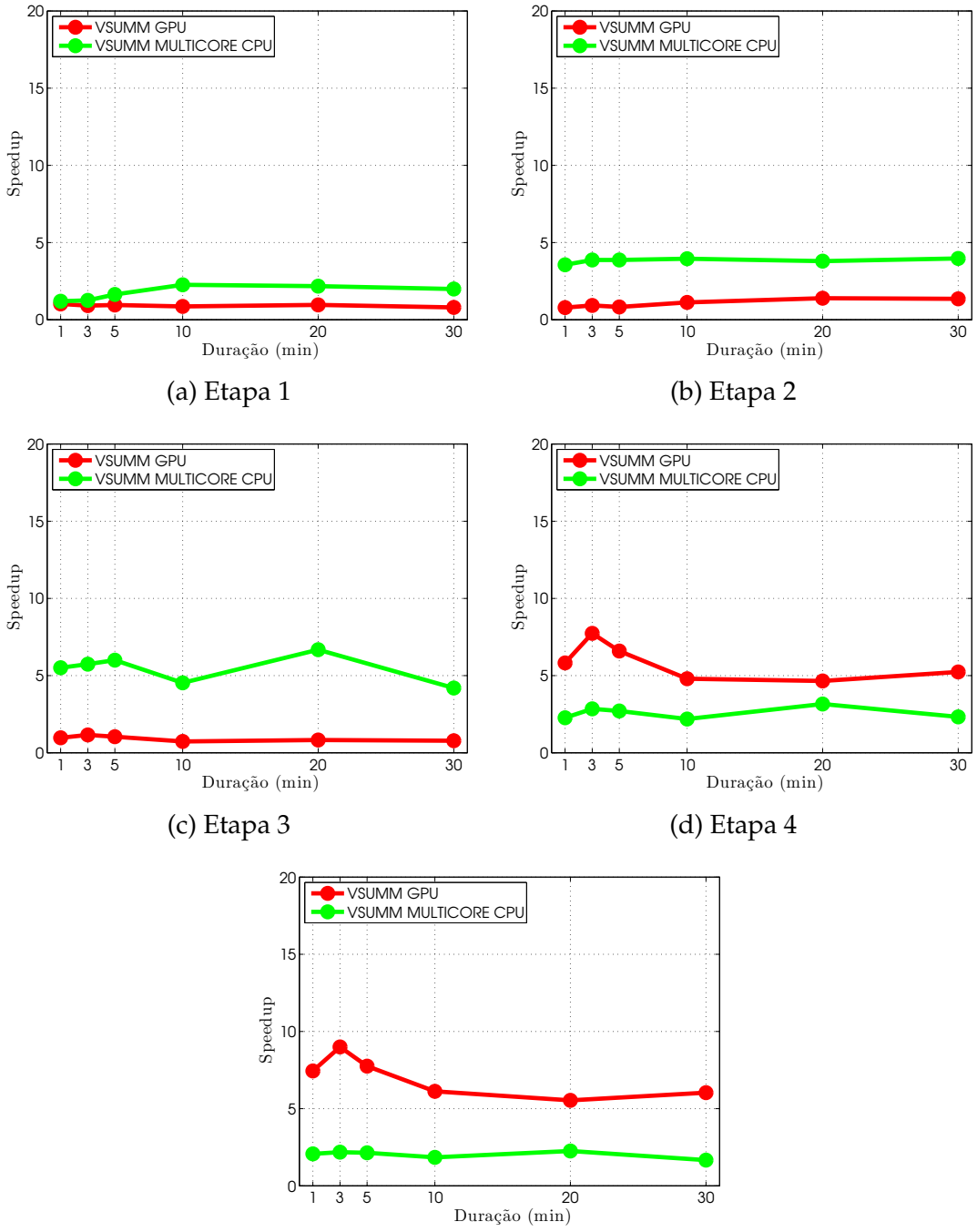
Tabela 5.11: Resultados TEMPSUMM - tempo de execução e desvio padrão versão Multicore CPU

Resultados TEMPSUMM - Versão Multicore CPU							
Resolução	**Duração (min)**	**Etapa 1**	**Etapa 2**	**Etapa 3**	**Etapa 4**	**Etapa 5**	**Total**
320 × 240	1	0.678(0.62)	0.079(0.00)	0.919(0.26)	4.083(0.10)	0.132(0.04)	10.576(1.65)
320 × 240	3	1.453(0.66)	0.209(0.01)	2.988(1.18)	0.921(0.36)	0.414(0.16)	24.118(7.31)
320 × 240	5	1.625(0.59)	0.358(0.03)	2.466(1.59)	4.078(0.10)	0.359(0.25)	21.682(9.41)
320 × 240	10	2.244(0.58)	0.714(0.05)	5.981(3.42)	1.866(1.10)	0.858(0.52)	44.508(20.13)
320 × 240	20	4.941(0.89)	1.546(0.20)	10.908(7.26)	4.153(0.05)	1.531(1.07)	79.088(43.57)
320 × 240	30	8.227(2.04)	2.274(0.20)	17.987(12.24)	4.837(3.09)	2.381(1.55)	126.641(71.77)
640 × 360	1	0.809(0.63)	0.185(0.01)	0.974(0.29)	34.757(94.60)	0.346(0.09)	42.756(94.45)
640 × 360	3	1.975(0.74)	0.544(0.05)	3.236(1.25)	2.618(0.99)	1.183(0.50)	30.046(8.29)
640 × 360	5	2.307(0.74)	0.896(0.05)	3.039(1.57)	4.117(0.08)	1.081(0.52)	29.854(10.12)
640 × 360	10	3.358(0.81)	1.847(0.18)	6.444(3.12)	5.076(2.33)	2.365(1.25)	57.151(20.80)
640 × 360	20	10.565(4.98)	3.968(0.37)	12.090(8.16)	4.252(0.31)	4.281(3.09)	107.350(56.66)
640 × 360	30	17.032(5.52)	5.488(0.20)	19.998(13.65)	15.565(11.47)	7.542(5.50)	173.371(96.76)
1280 × 720	1	1.579(0.63)	0.678(0.04)	1.312(0.34)	180.788(407.11)	1.149(0.28)	196.661(406.90)
1280 × 720	3	3.945(0.65)	2.058(0.09)	4.475(1.90)	10.957(4.05)	3.640(1.54)	54.569(16.79)
1280 × 720	5	5.399(1.00)	3.401(0.24)	5.612(5.23)	4.037(0.06)	3.302(1.45)	64.801(30.85)
1280 × 720	10	14.649(4.72)	6.962(0.40)	8.604(4.16)	21.264(10.06)	7.776(3.99)	119.405(38.13)
1280 × 720	20	36.981(9.83)	14.382(1.42)	18.928(16.49)	4.061(0.10)	13.991(10.07)	244.212(125.92)
1280 × 720	30	57.469(9.31)	20.692(0.70)	27.249(14.97)	60.713(34.75)	22.647(13.68)	367.136(129.45)
1920 × 1080	1	2.833(0.64)	1.437(0.08)	1.843(0.42)	4.063(0.09)	2.345(0.49)	31.937(4.97)
1920 × 1080	3	7.112(0.73)	4.389(0.15)	6.120(2.15)	22.340(9.15)	7.732(2.81)	91.562(23.47)
1920 × 1080	5	12.572(2.56)	7.437(0.46)	5.256(2.24)	4.082(0.14)	6.493(2.91)	96.292(24.01)
1920 × 1080	10	30.387(5.72)	15.266(0.98)	11.501(5.81)	41.858(21.72)	15.225(8.16)	207.569(64.27)
1920 × 1080	20	77.001(21.60)	31.479(2.73)	268.179(781.94)	4.111(0.13)	29.260(19.95)	1768.303(4316.03)
1920 × 1080	30	96.133(12.89)	45.136(1.55)	29.917(21.05)	109.107(88.46)	39.814(30.38)	566.436(250.78)

Nas implementações em GPU da abordagem VSUMM, chegou-se à conclusão de que não é interessante para GPU copiar sucessivos quadros da CPU para GPU, ao invés de copiar um conjunto grande de quadros de acordo com a memória e limitações da GPU. Dessa forma, a tabela 5.12 apresenta os resultados obtidos pela versão GPU em que a decodificação é realizada em CPU e não em GPU. Esses dados mostram que as etapas 4 e 5 foram as mais aceleradas na versão GPU.

Para comprovar qual versão obteve melhores resultados em cada etapa do TEMPSUMM, ambas foram comparadas com a versão sequencial em relação ao *speedup*, como apresentado nos gráficos da Figuras 5.5, 5.6, 5.7 e 5.8.

Os resultados de *speedup* mostram que a decodificação utilizando *multicore* CPU é mais rápida do que a implementação sequencial, também utilizada na versão GPU. O resultado foi inferior ao obtido pela decodificação do VSUMM pois neste caso foram decodificados todos os quadros do vídeo e não apenas um por segundo, resultando em uma diminuição do *speedup*. A etapa de segmentação do vídeo também foi mais rápido na versão *multicore* CPU, chegando a um *speedup* de aproximadamente 5×. Novamente, a versão *multithread* apresentou os melhores resultados também para etapa de detecção de quadros representativos, com *speedup* médio de 5×.



(a) Etapa 1

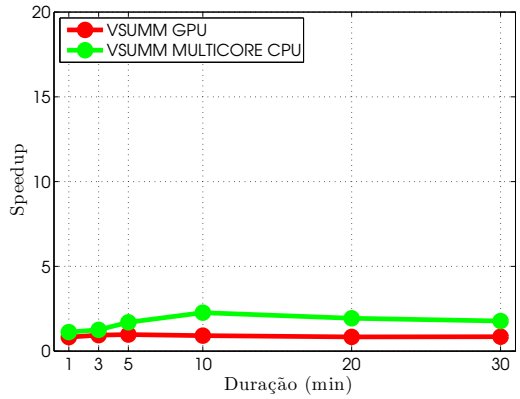
(b) Etapa 2

(c) Etapa 3

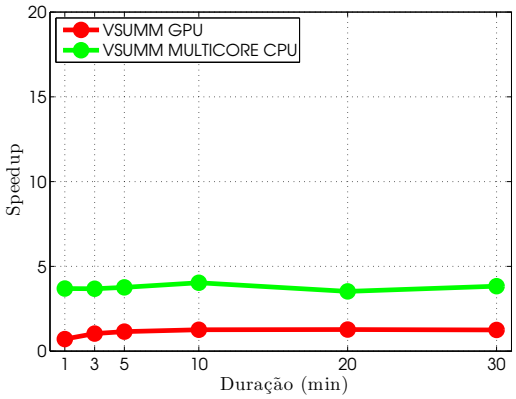
(d) Etapa 4

(e) Etapa 5

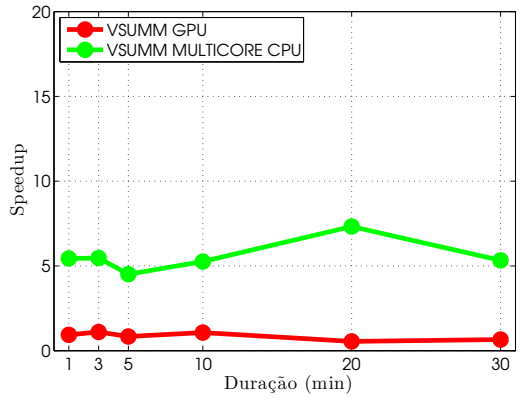
Figura 5.5: *Speedups* obtidos pelas versões *multicore* CPU e GPU nas etapas 1 a 5 com os vídeos resolução 320 × 240



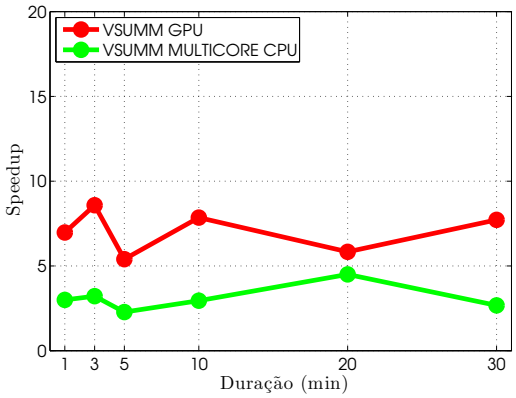
(a) Etapa 1



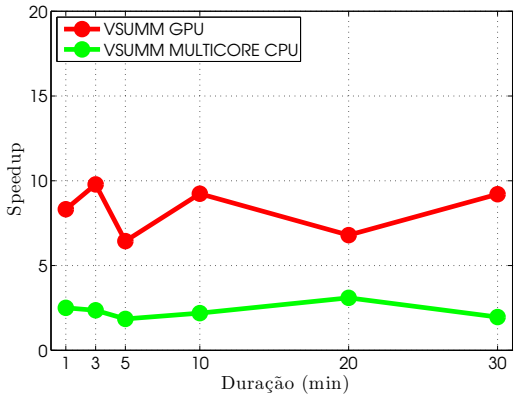
(b) Etapa 2



(c) Etapa 3

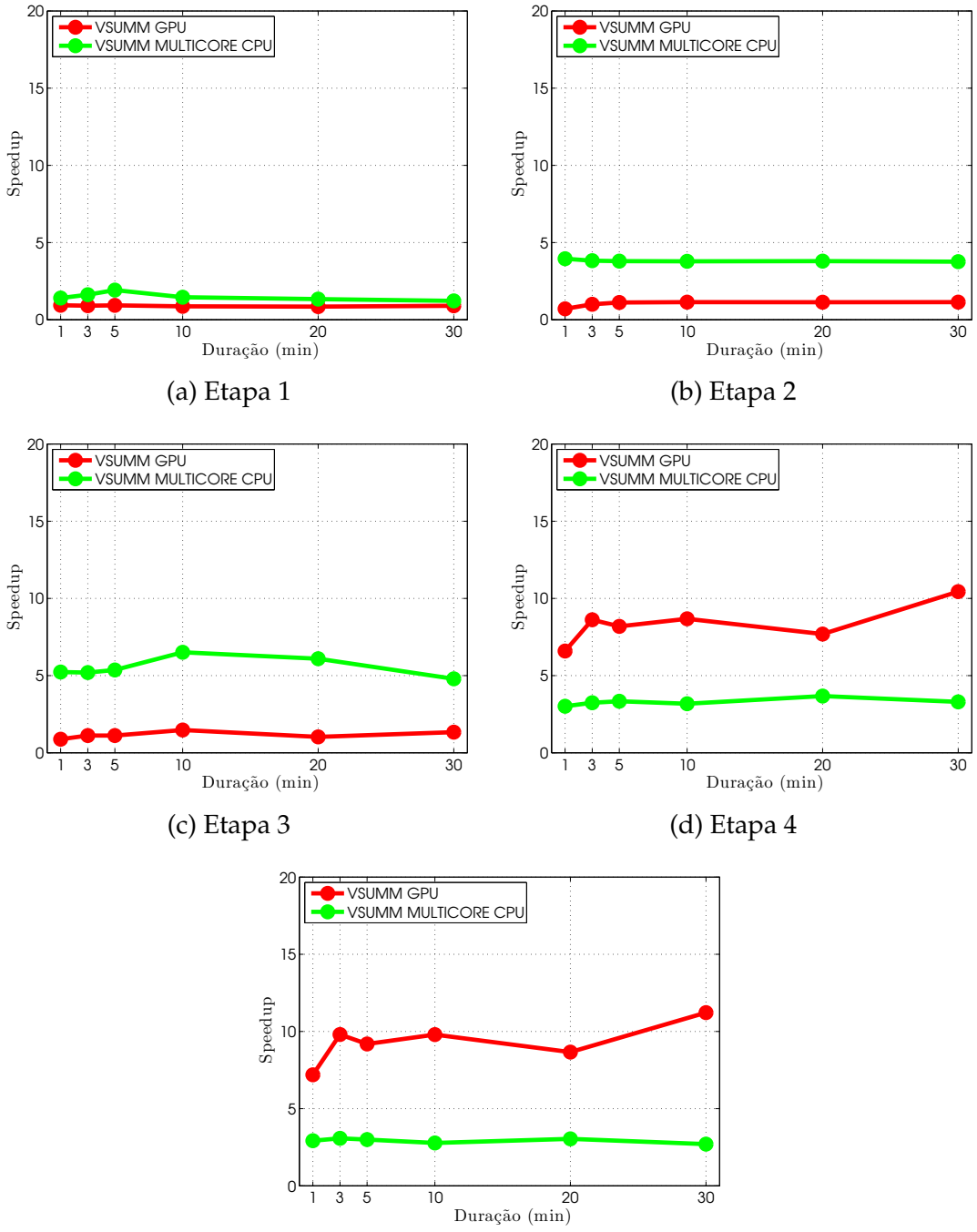


(d) Etapa 4



(e) Etapa 5

Figura 5.6: *Speedups* obtidos pelas versões *multicore* CPU e GPU nas etapas 1 a 5 com os vídeos resolução 640 × 360



(a) Etapa 1

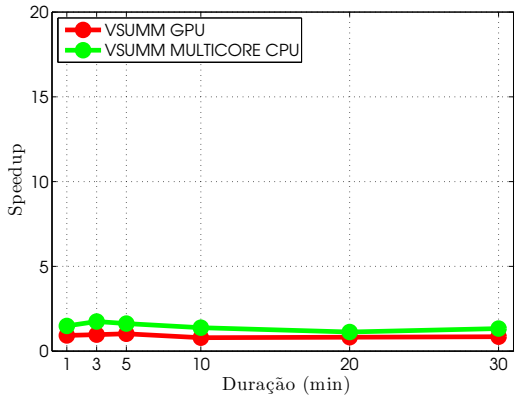
(b) Etapa 2

(c) Etapa 3

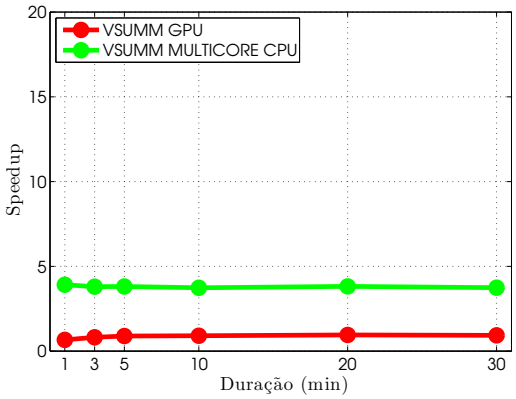
(d) Etapa 4

(e) Etapa 5

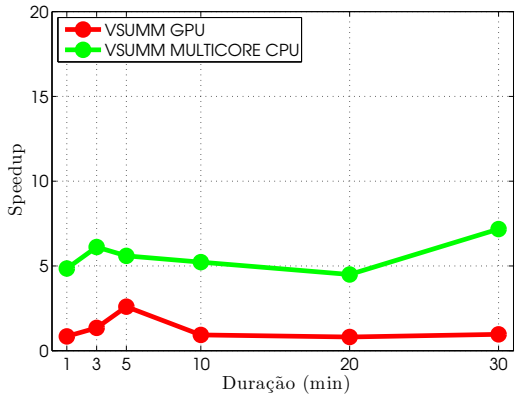
Figura 5.7: *Speedups* obtidos pelas versões *multicore* CPU e GPU nas etapas 1 a 5 com os vídeos resolução 1280×720



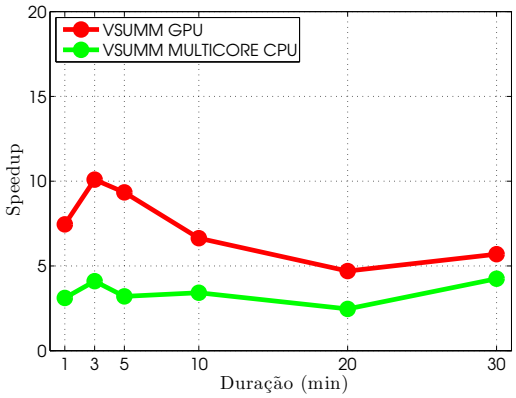
(a) Etapa 1



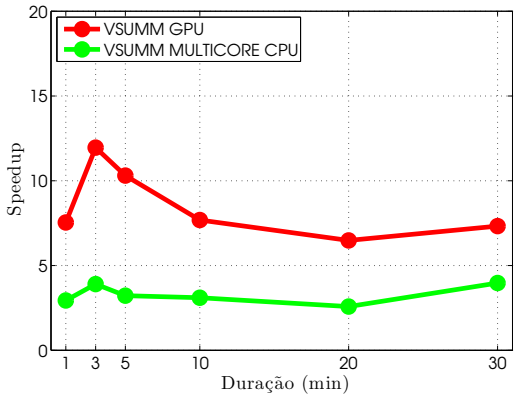
(b) Etapa 2



(c) Etapa 3



(d) Etapa 4



(e) Etapa 5

Figura 5.8: *Speedups* obtidos pelas versões *multicore* CPU e GPU nas etapas 1 a 5 com os vídeos resolução 1920×1080

Tabela 5.12: Resultados TEMPSUMM - tempo de execução e desvio padrão versão GPU

Resultados TEMPSUMM - Versão GPU							
Resolução	**Duração (min)**	**Etapa 1**	**Etapa 2**	**Etapa 3**	**Etapa 4**	**Etapa 5**	**Total**
320 × 240	1	0.792(0.36)	0.184(0.01)	5.240(1.65)	4.457(1.42)	0.037(0.01)	10.993(1.97)
320 × 240	3	2.000(0.39)	0.453(0.26)	14.847(7.17)	0.339(0.14)	0.100(0.05)	22.178(7.33)
320 × 240	5	2.849(0.38)	0.785(0.36)	36.616(69.76)	7.159(7.59)	0.107(0.08)	48.274(68.76)
320 × 240	10	5.911(1.20)	1.295(0.39)	36.893(17.20)	0.850(0.39)	0.258(0.12)	50.240(17.81)
320 × 240	20	13.336(4.95)	2.428(0.39)	246.692(588.32)	121.312(238.14)	0.423(0.28)	387.141(587.65)
320 × 240	30	20.681(5.09)	3.442(0.47)	154.847(198.47)	1.971(1.33)	0.605(0.41)	188.298(197.25)
640 × 360	1	1.098(0.49)	0.503(0.02)	5.623(1.94)	4.019(0.06)	0.100(0.03)	12.060(2.06)
640 × 360	3	2.633(0.41)	1.002(0.04)	16.004(7.53)	0.981(0.52)	0.285(0.15)	25.985(8.57)
640 × 360	5	4.006(0.33)	1.517(0.07)	16.431(10.16)	7.934(9.16)	0.311(0.19)	32.395(12.64)
640 × 360	10	8.352(1.19)	3.051(0.33)	31.672(17.05)	1.906(1.18)	0.561(0.35)	52.169(19.48)
640 × 360	20	23.584(4.85)	6.147(0.57)	106.405(152.30)	48.410(92.41)	1.161(0.94)	193.606(155.34)
640 × 360	30	35.721(5.27)	8.715(0.40)	162.402(152.35)	5.382(4.18)	1.606(1.24)	224.149(153.97)
1280 × 720	1	2.377(0.44)	1.929(0.06)	7.800(2.58)	4.049(0.06)	0.465(0.15)	19.902(3.51)
1280 × 720	3	7.030(0.94)	4.060(0.23)	20.854(9.87)	4.120(2.10)	1.142(0.59)	45.301(14.53)
1280 × 720	5	11.526(2.12)	6.106(0.45)	19.367(11.31)	24.621(65.03)	1.078(0.63)	70.435(64.21)
1280 × 720	10	24.685(4.46)	11.956(0.48)	38.069(23.81)	7.774(5.15)	2.205(1.46)	97.045(36.97)
1280 × 720	20	54.688(6.43)	25.044(2.43)	88.377(49.81)	5.080(2.29)	4.665(3.26)	210.474(70.24)
1280 × 720	30	77.861(10.07)	35.354(0.99)	97.765(86.61)	19.171(17.48)	5.454(4.94)	259.972(134.47)
1920 × 1080	1	4.572(0.55)	4.399(0.25)	10.685(2.92)	12.458(26.63)	0.915(0.21)	39.534(26.21)
1920 × 1080	3	12.772(0.94)	10.604(0.29)	29.730(13.63)	9.530(4.49)	2.684(1.28)	115.250(114.82)
1920 × 1080	5	20.399(1.68)	16.685(0.63)	21.108(15.69)	5.117(3.62)	1.742(1.45)	77.722(29.01)
1920 × 1080	10	49.807(8.34)	32.872(1.25)	53.637(29.72)	17.684(10.38)	5.061(2.88)	1575.436(2932.47)
1920 × 1080	20	106.287(10.90)	64.976(5.02)	123.245(52.33)	4.023(0.08)	11.959(5.24)	395.017(93.91)
1920 × 1080	30	146.950(10.38)	94.261(2.05)	175.991(90.03)	60.073(32.32)	16.690(8.94)	2717.194(6739.63)

As etapas da sumarização em que ocorrem a clusterização utilizando o algoritmo LBG e o cálculo do histograma de palavras visuais foram aceleradas de forma mais efetiva com a versão em GPU. Os *speedups* relacionados à etapa 4 variaram entre 5× e 10×. Já o cálculo do histograma de palavras visuais obteve *speedup* superior 10× em alguns casos, principalmente nos vídeos de resoluções maiores.

5.3.3 Implementação e Experimentos Versão Híbrida

A seguir são apresentadas as versões escolhidas para cada etapa da versão híbrida, juntamente com a explicação dessas escolhas, também apresentadas na Figura 6.

5.3.3.1 Implementação Versão Híbrida

- Decodificação do Vídeo: *multicore* CPU. A decodificação do vídeo realizada nesta abordagem é a mesma utilizada para a abordagem VSUMM, diferenciando apenas na quantidade de quadros é decodificada por segundo. Dessa forma, a versão *multicore* CPU apresentou os melhores resultados.

- Segmentação Temporal do Vídeo: *multicore* CPU. Como a maioria dos resulta-

dos mais ágeis desta etapa foram utilizando *multicore* CPU, esta foi a escolha para versão híbrida, mesmo com os vídeos maiores apresentando melhores resultados em GPU.

- Detecção de Quadros Representativos: *multicore* CPU. Essa etapa é realizada pelo algoritmo x-means, onde foi utilizada a versão executável fornecida pelos autores do algoritmo. Então os CUDA cores são conseguem rodar um arquivo executável como os cores da CPU. Dessa forma a versão *multicore* CPU apresentou resultados mais ágeis do que a versão sequencial.

- *Bag of Visual Words*: GPU. Os experimentos mostraram que esta etapa apresentou resultados muito superiores com a implementação em GPU, principalmente para a clusterização com o método LBG.

- Histograma de *Visual Words*: GPU. Novamente, os diversos CUDA cores da GPUs apresentaram resultados mais eficientes para o cálculo do histograma de *visual words* em relação à outra versão paralela implementada.

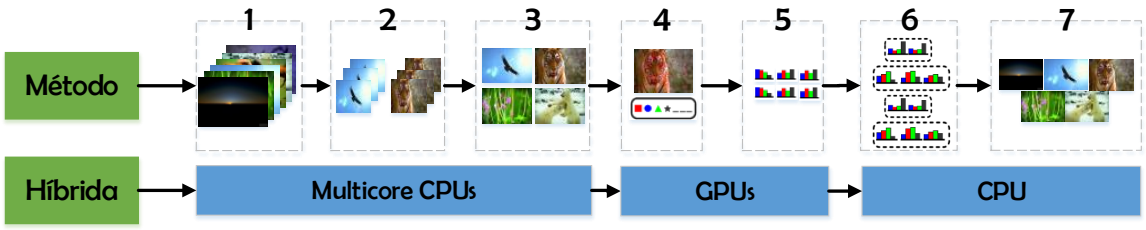


Figura 5.9: Diagrama de Fluxo de Dados/Execução da Implementação Híbrida para a Abordagem de Sumarização de Vídeos proposta em Cahuina et al. [2013]

Dada essa versão híbrida, foram realizados testes para comprovar que ela é melhor que as duas outras versões implementadas. Nesses testes, como nos do VSUMM, foi medida a quantidade mínima, máxima e média de quadros processados por segundo (FPS) em todas as implementações. Além disso, essa métrica foi analisada por gênero, duração e resolução. As tabelas 5.13, 5.14 e 5.15 mostram os resultados desses experimentos. É possível observar que a quantidade de quadros processados por segundo com esse método é menor que do com o VSUMM, pois nesse caso, são processados todos os quadros do vídeo e não apenas um quadro por segundo. Esse fato também influencia no processamento de várias etapas.

Em relação aos gêneros dos vídeos, a Tabela 5.13 mostra que os resultados são variaram bastante para os diferentes tipos de vídeos. O método de sumarização

Tabela 5.13: Efetividade em quadros processados por segundo para as versões implementadas em relação aos gêneros dos vídeos.

Resultados TEMPSUMM - Quadros Processados por Segundo por Gênero				
Versão	**Gênero**	**FPS Min**	**FPS Max**	**FPS Médio**
CPU	Animais	0.00	13.00	4.33
CPU	Carro	0.00	22.00	5.83
CPU	Desenho	1.00	23.00	6.33
CPU	Comédia	0.00	49.00	9.83
CPU	Entrevista	0.00	38.00	10.46
CPU	Filme	1.00	23.00	7.00
CPU	Música	0.00	11.00	3.71
CPU	Natureza	1.00	21.00	5.54
CPU	Jornal	1.00	33.00	9.75
CPU	Esporte	0.00	7.00	2.96
GPU	Animais	2.00	14.00	6.17
GPU	Carro	2.00	32.00	7.21
GPU	Desenho	2.00	18.00	7.46
GPU	Comédia	0.00	35.00	8.00
GPU	Entrevista	2.00	30.00	12.21
GPU	Filme	0.00	28.00	8.88
GPU	Música	0.00	13.00	6.21
GPU	Natureza	1.00	11.00	5.17
GPU	Jornal	2.00	35.00	9.62
GPU	Esporte	1.00	11.00	4.75
Multicore CPU	Animais	0.00	12.00	5.29
Multicore CPU	Carro	2.00	23.00	6.83
Multicore CPU	Desenho	2.00	25.00	7.50
Multicore CPU	Comédia	2.00	57.00	11.75
Multicore CPU	Entrevista	0.00	45.00	13.54
Multicore CPU	Filme	1.00	25.00	8.54
Multicore CPU	Música	1.00	15.00	5.79
Multicore CPU	Natureza	2.00	21.00	6.92
Multicore CPU	Jornal	0.00	38.00	11.75
Multicore CPU	Esporte	1.00	17.00	5.42
Híbrida	Animais	2.00	16.00	7.85
Híbrida	Carro	2.00	32.00	7.97
Híbrida	Desenho	2.00	27.00	8.10
Híbrida	Comédia	2.00	65.00	13.58
Híbrida	Entrevista	2.00	49.00	16.47
Híbrida	Filme	1.00	29.00	8.59
Híbrida	Música	1.00	16.00	6.92
Híbrida	Natureza	2.00	21.00	6.24
Híbrida	Jornal	2.00	41.00	14.28
Híbrida	Esporte	1.00	17.00	5.36

apresentou maior facilidade no processamento de vídeos onde não há muitas tomadas, onde os quadros são muito parecidos, como nos vídeos de entrevista, jornal e comédia. Já nos gêneros mais agitados e com grandes movimentações como esportes e música o TEMPSUMM obteve resultados inferiores. Em vários casos, o FPS mínimo foi de zero, ou seja, em um segundo o método não conseguiu processar nenhum quadro completo, foi necessário mais de um segundo.

Analisando a quantidade de quadros processados por segundo com relação à duração dos vídeos, a Tabela 5.14 apresenta um resultado diferente do apresentado pelo VSUMM. Nesse caso, a quantidade de quadros processados por segundo

Tabela 5.14: Efetividade em quadros processados por segundo para as versões implementadas em relação à duração dos vídeos.

Resultados TEMPSUMM - Quadros Processados por Segundo por Duração				
Versão	**Duração**	**FPS Min**	**FPS Max**	**FPS Médio**
CPU	1	0.00	6.00	3.20
CPU	3	1.00	11.00	4.05
CPU	5	0.00	23.00	7.38
CPU	10	0.00	35.00	7.70
CPU	20	0.00	49.00	9.30
CPU	30	0.00	38.00	7.83
GPU	1	1.00	7.00	3.88
GPU	3	0.00	15.00	5.80
GPU	5	1.00	24.00	8.38
GPU	10	0.00	32.00	9.25
GPU	20	1.00	35.00	7.85
GPU	30	0.00	35.00	10.25
Multicore CPU	1	0.00	7.00	3.62
Multicore CPU	3	1.00	12.00	5.10
Multicore CPU	5	2.00	25.00	9.00
Multicore CPU	10	2.00	38.00	9.68
Multicore CPU	20	0.00	57.00	11.70
Multicore CPU	30	2.00	45.00	10.90
Híbrida	1	1.00	7.00	3.92
Híbrida	3	1.00	16.00	6.16
Híbrida	5	2.00	27.00	9.20
Híbrida	10	2.00	42.00	9.98
Híbrida	20	1.00	57.00	12.35
Híbrida	30	2.00	48.00	11.40

foi proporcional à duração do vídeo. Isso mostra que a duração é um fator que influencia nos resultados do TEMPSUMM, e ocorre uma maior facilidade do processamento de muitos quadros dos vídeos de maior duração.

Os resultados apresentados na Tabela 5.15 mostram que a resolução é um fator que também influencia muito o FPS mínimo, máximo e médio. O FPS muda muito de uma resolução para outra e isto ocorre de maneira inversamente proporcional à resolução, ou seja, quanto maior a resolução, menos quadros são processados por segundo.

Como esperado, a versão híbrida atingiu os maiores FPS em todos os gêneros, durações e resoluções, chegando a um FPS máximo de 65.

5.4 Análise dos Resultados Obtidos

Os resultados obtidos mostram a eficiência da utilização de paralelismo para a sumarização automática de vídeos. Tanto as versões utilizando *multicore* CPU quanto as que utilizaram GPUs apresentaram resultados satisfatórios para as duas abordagens de sumarização analisadas, e com os dois computadores testados. Como

Tabela 5.15: Efetividade em quadros processados por segundo para as versões implementadas em relação à resolução dos vídeos.

Resultados TEMPSUMM - Quadros Processados por Segundo por Resolução				
Versão	**Resolução**	**FPS Min**	**FPS Max**	**FPS Médio**
CPU	320 × 240	0.00	49.00	12.25
CPU	640 × 360	0.00	26.00	8.38
CPU	1280 × 720	1.00	13.00	3.65
CPU	1920 × 1080	0.00	6.00	2.02
GPU	320 × 240	1.00	35.00	11.27
GPU	640 × 360	2.00	28.00	10.22
GPU	1280 × 720	1.00	13.00	5.87
GPU	1920 × 1080	0.00	6.00	2.92
Multicore CPU	320 × 240	4.00	57.00	14.80
Multicore CPU	640 × 360	0.00	36.00	10.82
Multicore CPU	1280 × 720	0.00	13.00	4.83
Multicore CPU	1920 × 1080	0.00	7.00	2.88
Híbrida	320 × 240	4.00	65.00	18.45
Híbrida	640 × 360	2.00	39.00	11.82
Híbrida	1280 × 720	1.00	13.00	5.97
Híbrida	1920 × 1080	0.00	7.00	2.93

esperado, a GPU apresentou melhores resultados nas etapas que possuem muitos cálculos matemáticos repetitivos. Os milhares de CUDA cores têm a habilidade de realizar esses cálculos simultaneamente e de forma eficiente. Além disso, a diferença na quantidade de CUDA cores e de memória das GPUs dos dois computadores analisados mostrou grande influência no *speedup* obtidos para as etapas dos algoritmos. Por outro lado, em tarefas que envolveram utilização de funções de bibliotecas, as versões *multicore* CPU atingiram os melhores resultados. Isso ocorreu pois as GPUs não executam esses tipos de funções, apenas cálculos simples.

Entre as duas abordagens de sumarização analisadas, a primeira é muito mais simples e rápida que a segunda. Comparando a qualidade dos resumos gerados, Cahuina et al. [2013] mostrou que seus resultados são melhores porém com uma diferença mínima em relação aos resultados da abordagem apresentada em Avila et al. [2011]. No entanto, essa melhoria não se mostrou significativa e comparando o tempo de execução, mesmo nas versões paralelas, utilizar a abordagem VSUMM é mais vantajoso.

Em relação à qualidade dos resumos, como houve uma preocupação em manter todos os detalhes e características dos métodos originais, nesse trabalho não foi medida essa qualidade. Como os métodos paralelos apresentam os mesmos resultados dos métodos sequenciais, a qualidade dos resumos é mantida. Dessa forma, a qualidade dos resumos gerados nesse trabalho é a mesma apresentada nos trabalhos utilizados como base (Avila et al. [2011]; Cahuina et al. [2013]).

5.5 Discussão sobre uma Implementação Paralela Genérica para Sumarização de Vídeos

Foram apresentadas implementações paralelas em GPU e *multicore* CPUs, especificamente, para dois métodos de sumarização automática de vídeos, propostos em Avila et al. [2011] e Cahuina et al. [2013]. Além disso, este capítulo apresentou uma versão híbrida dessas abordagens, sendo essa a união das etapas das versões anteriores que apresentaram os melhores resultados. No entanto, de acordo com Truong & Venkatesh [2007], existem alguns aspectos fundamentais presentes em diferentes métodos de sumarização de vídeos para geração de resumos estáticos, como apresentado na Figura 5.10.

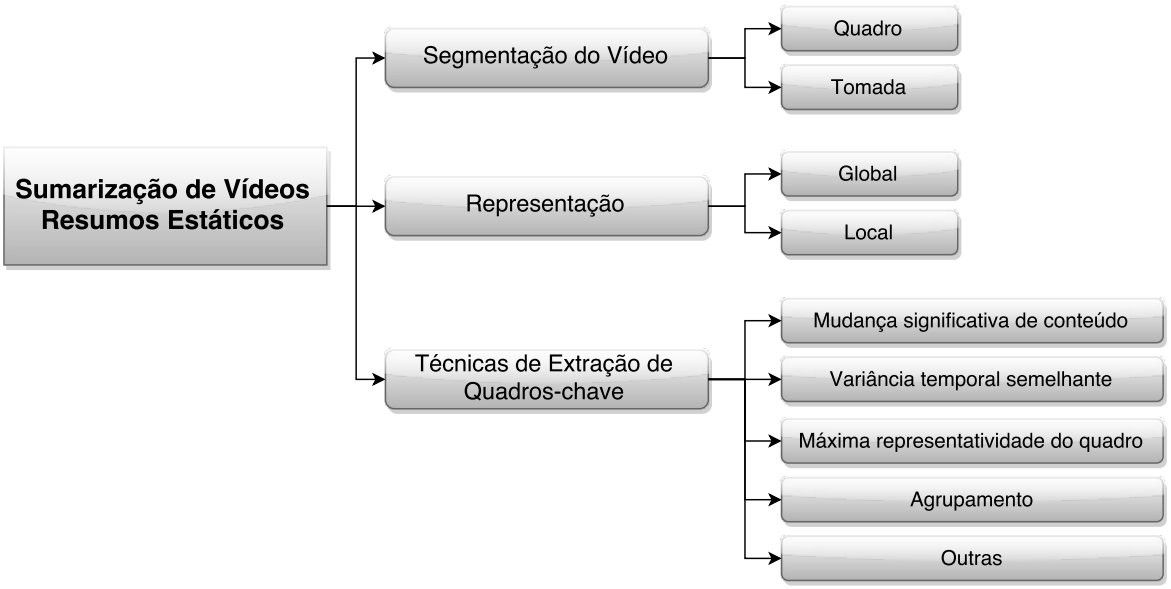


Figura 5.10: Alguns atributos fundamentais de métodos de sumarização de vídeos descritos em Truong & Venkatesh [2007].

Inicialmente o vídeo é segmentado em quadros ou tomadas. Então são extraídas características globais ou locais para representação dos quadros. A partir dessas características, os quadros-chave são extraídos utilizando alguma técnica de extração. Esses atributos ocorrem na maioria dos métodos de sumarização da literatura.

Dessa forma, as ideias de algoritmos paralelos apresentadas anteriormente podem ser aplicadas sem grandes modificações para diferentes abordagens de sumarização de vídeos que seguem este mesmo padrão. Em especial, tomando como base a versão híbrida, que apresentou os melhores resultados, é possível desenvol-

ver uma versão paralela eficiente para outros métodos de sumarização de vídeos. Ou seja, a segmentação dos quadros é a mesma realizada por todos os algoritmos e implementada nesse trabalho. Nas partes onde várias computações são realizadas igualmente em todos os quadros, como na representação dos quadros, basta utilizar um *grid* CUDA tridimensional e implementar o *kernel* com essas computações. A identificação dos quadros-chave nem sempre é paralelizável, porém se o algoritmo percorrer os quadros comparando-os com outros ou uma tarefa similar, é possível realizar essas comparações em GPU.

Capítulo 6

Conclusões e Trabalhos Futuros

Os avanços em técnicas de compressão, a redução do custo de armazenagem e a disponibilidade de conexões de alta velocidade facilitaram a criação, armazenamento e distribuição de vídeos. Com isso, a quantidade de vídeos gerada em diversas aplicações aumentou significantemente, introduzindo a necessidade de um gerenciamento mais eficiente destes vídeos. A sumarização automática de vídeos é uma forma de realizar este gerenciamento, no entanto, a maioria dos métodos que realizam essa tarefa demoram muito tempo para executar, o que é indesejável. Uma forma de reduzir este tempo de execução é utilizar as vantagens das recentes arquiteturas de computadores que permitem alto grau de paralelismo, como as unidades de processamento gráficos (GPUs) e unidade central de processamento com múltiplos núcleos (*multicore* CPUs).

Com base nestas informações, este trabalho propôs implementações paralelas para dois algoritmos de sumarização automática de vídeos, apresentados em Avila et al. [2011] e Cahuina et al. [2013]. A primeira implementação utiliza *multicore* CPUs e apresentou resultados eficientes para decodificação de vídeos. A segunda utiliza GPUs, mostrando-se mais eficiente nas etapas que são constituídas por muitas computações matemáticas simples. Finalmente, a terceira versão, chamada híbrida, combina as melhores partes das versões anteriores.

Para avaliar a eficiência das implementações propostas, foi criada um conjunto de dados com 240 vídeos. Esses vídeos possuem diferentes resoluções, tamanhos e são de diferentes gêneros. Os experimentos realizados relataram detalhadamente o tempo de execução e *speedup* de todas as etapas dos dois algoritmos em estudo. Além disso, foi medida a quantidade de quadros que as versões conseguiram processar por segundo. Por fim, foram apresentadas diversas análises dos resultados, bem como comparações entre os dois métodos e a possibilidade de utilizar as ideias

propostas em outros algoritmos de sumarização automática de vídeos.

Os experimentos mostraram que as implementações paralelas, tanto *multicore* CPU quanto GPU reduziram de forma significativa o tempo de execução do algoritmo. A versão híbrida acelerou ainda mais, atingindo os melhores resultados. Para essa versão, o método de sumarização VSUMM alcançou em média 5× de *speedup*, sendo que, para a etapa de extração de características, o *speedup* chegou a 20×. A abordagem de sumarização de vídeos proposta por Cahuina et al. [2013] atingiu *speedup* médio de 4×, enquanto o *speedup* das etapas separadamente variou entre 3× e 10×.

Com relação aos trabalhos futuros, o presente trabalho pode ser continuado explorando-se os seguintes aspectos:

- Estudo sobre modificações de etapas e algoritmos nas versões paralelas para que não sejam tão fiéis aos algoritmos originais mas que obtenham resultado similiar e com mais rapidez.
- Estudo sobre a utilização de *Cloud Computing* nas implementações paralelas, também com o objetivo de deixá-las mais ágeis.

6.1 Publicações

Conferências Internacionais

- Almeida, S. S.; Nazaré Júnior, A. C.; de A. Araújo, A.; Cámara-Chávez, G. & Menotti, D.. Speeding up a video summarization approach using GPUs and multicore CPUs. Em Procedia Computer Science of International Conference on Computational Science (ICCS), 29(0):159–171, Cairns, Austrália, 2014.
- Almeida, S. S.; Cahuina, E.; de A. Araújo, A.; Cámara-Chávez, G. & Menotti, D.. GPUs and Multicore CPUs Implementations of a Static Video Summarization. Em 19th Iberoamerican Congress on Pattern Recognition (CIARP), Puerto Vallarta, México, 2014.

Referências Bibliográficas

Almeida, J.; Leite, N. J. & Torres, R. d. S. (2012). Vison: Video summarization for online applications. *Pattern Recogn. Lett.*, 33(4):397--409. 1, 26, 28

Almeida, S. S. (2014). Implementações paralelas para algoritmos de sumarização de vídeos. https://github.com/susilvaalmeida/. 33

Almeida, S. S.; de Nazaré Júnior, A. C.; Araújo, A. d. A.; Cámara-Chávez, G. & Menotti, D. (2014). Speeding up a video summarization approach using GPUs and multicore CPUs. *Procedia Computer Science*, 29(0):159 – 171. 2014 International Conference on Computational Science. 1

Amdahl, G. M. (1967). Validity of the single processor approach to achieving large scale computing capabilities. Em *Proceedings of the April 18-20, 1967, Spring Joint Computer Conference*, AFIPS '67 (Spring), pp. 483--485, New York, NY, USA. ACM. 23

Avila, S. E. F.; ao Lopes, A. P. B.; da Luz Jr., A. & de Albuquerque Araújo, A. (2011). Vsumm: A mechanism designed to produce static video summaries and a novel evaluation method. *Pattern Recognition Letters*, 32(1):56 – 68. Image Processing, Computer Vision and Pattern Recognition in Latin America. xvii, xviii, xxi, xxii, 1, 2, 3, 4, 7, 8, 9, 10, 26, 27, 28, 33, 34, 35, 37, 39, 47, 48, 49, 51, 53, 55, 56, 57, 70, 71, 73

Avila, S. E. F. d. (2008). Uma abordagem baseada em características de cor para a elaboração automática e avaliação subjetiva de resumos estáticos de vídeos. Dissertação de mestrado, Universidade Federal de Minas Gerais. 10

Bay, H.; Ess, A.; Tuytelaars, T. & Gool, L. V. (2008). Speeded-up robust features (surf). *Computer Vision and Image Understanding*, 110(3):346 – 359. Similarity Matching in Computer Vision and Multimedia. 8, 13

Beaver, F. (2006). *Dictionary of Film Terms: The Aesthetic Companion to Film Art*. Peter Lang.

Cahuina, E. et al. (2013). A new method for static video summarization using local descriptors and video temporal segmentation. Em *Graphics, Patterns and Images (SIBGRAPI), 2013 26th SIBGRAPI - Conference on*, pp. 226–233. xvii, xviii, xxi, xxii, 1, 2, 3, 4, 7, 8, 11, 12, 27, 28, 33, 40, 41, 43, 45, 47, 59, 61, 63, 65, 67, 70, 71, 73, 74

Cahuina, E. J. Y. C. (2012). Static Video Summarization Based on Local Descriptors and Temporal Segmentation. Dissertação de mestrado, Federal University of Ouro Preto.

Cámara-Chávez, G.; Precioso, F.; Cord, M.; Phillip-Foliguet, S. & de AAraujo, A. (2007). Shot boundary detection by a hierarchical supervised approach. Em *Systems, Signals and Image Processing, 2007 and 6th EURASIP Conference focused on Speech and Image Processing, Multimedia Communications and Services. 14th International Workshop on*, pp. 197–200. 8, 13

Cheung, N.-M.; Fan, X.; Au, O. & Kung, M.-C. (2010). Video coding on multicore graphics processors. *Signal Processing Magazine, IEEE*, 27(2):79–89. 29

Clemons, J.; Jones, A.; Perricone, R.; Savarese, S. & Austin, T. (2011). Effex: An embedded processor for computer vision based feature extraction. Em *Design Automation Conference (DAC), 2011 48th ACM/EDAC/IEEE*, pp. 1020–1025. 29

Dalal, N. & Triggs, B. (2005). Histograms of oriented gradients for human detection. Em *Computer Vision and Pattern Recognition, 2005. CVPR 2005. IEEE Computer Society Conference on*, volume 1, pp. 886–893 vol. 1. 8, 13, 44

Diaz, J.; Munoz-Caro, C. & Nino, A. (2012). A survey of parallel programming models and tools in the multi and many-core era. *Parallel and Distributed Systems, IEEE Transactions on*, 23(8):1369–1386. 14, 15, 17

Ejaz, N. & Baik, S. (2012). Video summarization using a network of radial basis functions. *Multimedia Systems*, 18(6):483–497. 26, 28

Evangelio, R.; Senst, T.; Keller, I. & Sikora, T. (2013). Video indexing and summarization as a tool for privacy protection. Em *Digital Signal Processing (DSP), 2013 18th International Conference on*, pp. 1–6. 1

Farias, R.; Farias, R.; Marroquim, R. & Clua, E. (2013). Parallel image segmentation using reduction-sweeps on multicore processors and gpus. Em *Graphics, Patterns and Images (SIBGRAPI), 2013 26th SIBGRAPI - Conference on*, pp. 139–146. 30

Flynn, M. (1972). Some computer organizations and their effectiveness. *Computers, IEEE Transactions on*, C-21(9):948–960. 15

Furini, M.; Geraci, F.; Montangero, M. & Pellegrini, M. (2007). Visto: Visual storyboard for web video browsing. Em *Proceedings of the 6th ACM International Conference on Image and Video Retrieval*, CIVR '07, pp. 635--642, New York, NY, USA. ACM. 3, 8, 25, 26, 27, 28

Gao, Y.; Wang, W.-B.; Yong, J.-H. & Gu, H.-J. (2009). Dynamic video summarization using two-level redundancy detection. *Multimedia Tools and Applications*, 42(2):233–250. 1

Geraci, F.; Pellegrini, M.; Pisati, P. & Sebastiani, F. (2006). A scalable algorithm for high-quality clustering of web snippets. Em *Proceedings of the 2006 ACM Symposium on Applied Computing*, SAC '06, pp. 1058--1062, New York, NY, USA. ACM. 25

Gianluigi, C. & Raimondo, S. (2006). An innovative algorithm for key frame extraction in video summarization. *Journal of Real-Time Image Processing*, 1(1):69–88. 1, 8

Guan, G.; Wang, Z.; Yu, K.; Mei, S.; He, M. & Feng, D. (2012). Video summarization with global and local features. Em *Multimedia and Expo Workshops (ICMEW), 2012 IEEE International Conference on*, pp. 570–575. 1

Holub, P.; Srom, M.; Pulec, M.; Matela, J. & Jirman, M. (2013). Gpu-accelerated {DXT} and {JPEG} compression schemes for low-latency network transmissions of hd, 2k, and 4k video. *Future Generation Computer Systems*, 29(8):1991 – 2006. 30

Kasim, H.; March, V.; Zhang, R. & See, S. (2008). Survey on parallel programming model. Em Cao, J.; Li, M.; Wu, M.-Y. & Chen, J., editores, *Network and Parallel Computing*, volume 5245 of *Lecture Notes in Computer Science*, pp. 266–275. Springer Berlin Heidelberg. 17

Kuanar, S. K.; Panda, R. & Chowdhury, A. S. (2013). Video key frame extraction through dynamic delaunay clustering with a structural constraint. *Journal of Visual Communication and Image Representation*, 24(7):1212 – 1227. 27, 28

Li, P.; Sun, H.; Huang, C.; Shen, J. & Nie, Y. (2012). Interactive image/video retexturing using {GPU} parallelism. *Computers & Graphics*, 36(8):1048 – 1059. Graphics Interaction Virtual Environments and Applications 2012. 29

Li, Y.; Li, W. & Wang, H. (2013). Dynamic video summarization with content analysis. Em *Proceedings of the Fifth International Conference on Internet Multimedia Computing and Service*, ICIMCS '13, pp. 126--129, New York, NY, USA. ACM. 1

Linde, Y.; Buzo, A. & Gray, R. (1980). An algorithm for vector quantizer design. *Communications, IEEE Transactions on*, 28(1):84–95. 13

Lowe, D. (1999). Object recognition from local scale-invariant features. Em *Computer Vision, 1999. The Proceedings of the Seventh IEEE International Conference on*, volume 2, pp. 1150–1157 vol.2. 8, 13, 29

Macqueen, J. (1967). Some methods for classification and analysis of multivariate observations. Em *In 5-th Berkeley Symposium on Mathematical Statistics and Probability*, pp. 281--297. 8, 10, 13

Mundur, P.; Rao, Y. & Yesha, Y. (2006). Keyframe-based video summarization using delaunay clustering. *International Journal on Digital Libraries*, 6(2):219–232. 1, 3, 8, 25, 26, 27, 28

Navarro, C. a.; Hitschfeld-Kahler, N. & Mateu, L. (2013). A survey on parallel computing and its applications in data-parallel problems using gpu architectures. *Communications in Computational Physics*, 15(2):285--329. 14

NVIDIA (2014a). Cuda C Best Practices Guide. Relatório técnico February, NVIDIA. 23

NVIDIA (2014b). NVIDIA CUDA C Programming Guide. Relatório técnico, NVIDIA Corporation. xvii, 2, 19, 21

NVIDIA (2014c). NVIDIA CUDA Video Decoder. Relatório técnico February, NVIDIA. 35

OpenMP (2013). Openmp application program interface. Relatório técnico, NVIDIA Corporation. 17

OpenVideo (2011). The open video project. http://www.open-video.org. 25, 48

Palacios, J. & Triska, J. (2011). A comparison of modern gpu and cpu architectures: And the common convergence of both. Relatório técnico. 2

Pedronette, D.; Torres, R.; Borin, E. & Breternitz, M. (2013). Image re-ranking acceleration on gpus. Em *Computer Architecture and High Performance Computing (SBAC-PAD), 2013 25th International Symposium on*, pp. 176–183. 30

Pelleg, D. & Moore, A. W. (2000). X-means: Extending k-means with efficient estimation of the number of clusters. Em *Proceedings of the Seventeenth International Conference on Machine Learning*, ICML '00, pp. 727--734, San Francisco, CA, USA. Morgan Kaufmann Publishers Inc. 8, 13

Ryoo, S.; Rodrigues, C. I.; Baghsorkhi, S. S.; Stone, S. S.; Kirk, D. B. & Hwu, W.-m. W. (2008). Optimization principles and application performance evaluation of a multithreaded gpu using cuda. Em *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '08, pp. 73--82, New York, NY, USA. ACM. 20

Sanders, J. & Kandrot, E. (2010). *CUDA by Example: An Introduction to General-Purpose GPU Programming*. Addison-Wesley Professional, 1st edição. 37

Sena, M. C. R. & Costa, J. A. d. C. (2008). Tutorial openmp c/c++. Relatório técnico, LCCV-UFAL/Sun Microsystems do Brasil. 18

Sinha, S.; Frahm, J.-M.; Pollefeys, M. & Genc, Y. (2011). Feature tracking and matching in video using programmable graphics hardware. *Machine Vision and Applications*, 22(1):207–217. 28

Souza, T. T. P. (2013). Simulações Financeiras em GPU. Dissertação de mestrado, Universidade de São Paulo. 22

Tomasi, C. & Kanade, T. (1991). Detection and tracking of point features. Relatório técnico, International Journal of Computer Vision. 29

Truong, B. T. & Venkatesh, S. (2007). Video abstraction: A systematic review and classification. *ACM Trans. Multimedia Comput. Commun. Appl.*, 3(1). xviii, 1, 7, 71

Van Gemert, J.; Veenman, C.; Smeulders, A. W. M. & Geusebroek, J.-M. (2010). Visual word ambiguity. *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, 32(7):1271–1283. 13

Williams, A. (2012). *C++ Concurrency in Action: Practical Multithreading ; [for the New C++ 11 Standard]*. Manning Pubs Co Series. Manning Publications Company. 18

Won, J.-U.; Chung, Y.-S.; Kim, I.-S.; Choi, J.-G. & Park, K.-H. (2003). Correlation based video-dissolve detection. Em *Information Technology: Research and Education, 2003. Proceedings. ITRE2003. International Conference on*, pp. 104–107. 12

Wu, C.; Agarwal, S.; Curless, B. & Seitz, S. (2011). Multicore bundle adjustment. Em *Computer Vision and Pattern Recognition (CVPR), 2011 IEEE Conference on*, pp. 3057–3064. 29

Yang, J.; Jiang, Y.-G.; Hauptmann, A. G. & Ngo, C.-W. (2007). Evaluating bag-of-visual-words representations in scene classification. Em *Proceedings of the International Workshop on Workshop on Multimedia Information Retrieval*, MIR '07, pp. 197--206, New York, NY, USA. ACM. 13

Glossário

Cena é composta por um pequeno número de tomadas inter-relacionadas que são unificadas por características semelhantes e pela proximidade temporal. A união de todas as cenas forma o vídeo.
Centróide centro de grupos formados por algoritmos de agrupamento/clusterização.
Clusterização é uma técnica que consiste em agrupar um conjunto de padrões de acordo com uma medida de similaridade dos quais não se conhecem as informações das classes existentes.

Decodificação é a ação de transcrição, interpretação ou tradução de um código, de modo que possa ser entendido pelo decodificador ou seu utilizador. No caso de vídeo, a decodificação consiste em obter os quadros do vídeo na forma de imagens.
Descritor de Imagem descreve o conteúdo visual das imagens. Este conteúdo pode ser descrito através de informações dos objetos da imagem como cor, forma, textura, e normalmente é representado em vetores de características.

Histograma de Cor representa a distribuição da frequência de ocorrência dos valores cromáticos em uma imagem.

Quadro (frame) corresponde à imagem do vídeo, composto por *pixels* normalmente no formato RGB.
Quadro-chave (keyframe) é um quadro do vídeo que representa o conteúdo visual saliente de uma tomada. Dependendo da complexidade do conteúdo de uma tomada, um ou mais quadros-chave podem ser extraídos.
Quadros por Segundo é a taxa de quadros que define quantos quadros são mostrados a cada segundo do vídeo. Quanto maior a taxa de quadros, mais suave será a aparência do movimento durante a reprodução do vídeo.

Segmentação do Vídeo tem como objetivo separar o vídeo em seus componentes básicos, como em tomadas e em quadros.

Speedup é a razão entre o tempo de execução de um algoritmo de forma sequencial e o tempo de execução do mesmo algoritmo utilizando vários processadores.
Sumarização de Vídeo processo de extração de um resumo do conteúdo original do vídeo cujo objetivo é fornecer rapidamente uma informação concisa do seu conteúdo, preservando a mensagem original do vídeo.

Thread é uma forma de um processo dividir a si mesmo em duas ou mais tarefas que podem ser executadas concorrencialmente.
Tomada é uma sequência de quadros que apresenta uma ação contínua no tempo e no espaço que foi capturada por uma única câmera.

Vídeo Digital é uma sequência de quadros (imagens) em formato digital que, quando reproduzidos um a um, em determinada velocidade, apresentam a ilusão de movimento. Um vídeo pode ainda conter um canal de áudio sincronizado à sequência de imagens.