UNIVERSIDADE ESTADUAL PAULISTA "JÚLIO DE MESQUITA FILHO"
FACULDADE DE CIÊNCIAS - CAMPUS BAURU
DEPARTAMENTO DE COMPUTAÇÃO
BACHARELADO EM CIÊNCIA DA COMPUTAÇÃO

GUSTAVO HENRIQUE STAHL

PARALELIZAÇÃO DA TÉCNICA DE EXTRAÇÃO DE PONTOS DE INTERESSE FOAGDD UTILIZANDO A ARQUITETURA CUDA

BAURU
Janeiro/2023

GUSTAVO HENRIQUE STAHL

PARALELIZAÇÃO DA TÉCNICA DE EXTRAÇÃO DE PONTOS DE INTERESSE FOAGDD UTILIZANDO A ARQUITETURA CUDA

Trabalho de Conclusão de Curso do Curso de Ciência da Computação da Universidade Estadual Paulista "Júlio de Mesquita Filho", Faculdade de Ciências, Campus Bauru.
Orientador: Professor Associado Dr. Antonio Carlos Sementille

BAURU
Janeiro/2023

S781p

Stahl, Gustavo Henrique

Paralelização da técnica de extração de pontos de interesse FOAGDD utilizando a arquitetura CUDA / Gustavo Henrique Stahl. -- Bauru, 2023

58 p. : il., tabs., fotos

Trabalho de conclusão de curso (Bacharelado - Ciência da Computação) - Universidade Estadual Paulista (Unesp), Faculdade de Ciências, Bauru

Orientador: Antonio Carlos Sementille

1. Computação de alto desempenho. 2. NVIDIA CUDA. 3. Extrator de pontos de interesse. 4. Paralelismo. 5. Visão computacional. I. Título.

Sistema de geração automática de fichas catalográficas da Unesp. Biblioteca da Faculdade de Ciências, Bauru. Dados fornecidos pelo autor(a).

Essa ficha não pode ser modificada.

Gustavo Henrique Stahl

Paralelização da técnica de extração de pontos de interesse FOAGDD utilizando a arquitetura CUDA

Trabalho de Conclusão de Curso do Curso de Ciência da Computação da Universidade Estadual Paulista "Júlio de Mesquita Filho", Faculdade de Ciências, Campus Bauru.

Banca Examinadora

Professor Associado Dr. Antonio Carlos Sementille
Orientador
Universidade Estadual Paulista "Júlio de Mesquita Filho"
Faculdade de Ciências
Departamento de Ciência da Computação

Professora Dra. Simone das Graças Domingues Prado
Universidade Estadual Paulista "Júlio de Mesquita Filho"
Faculdade de Ciências
Departamento de Ciência da Computação

Professor Associado Dr. João Eduardo Machado Perea Martins
Universidade Estadual Paulista "Júlio de Mesquita Filho"
Faculdade de Ciências
Departamento de Ciência da Computação

Bauru, 24 de janeiro de 2023.

Dedico essa monografia a todos que buscam melhorar e expandir os limites na pesquisa científica.

Agradecimentos

Agradeço a minha namorada Isabela por ter me apoiado durante o desenvolvimento dessa pesquisa. Apesar de todas as dificuldades no caminho, ela estava sempre ao meu lado me incentivando a continuar. Sou grato também aos meus amigos, que nessa jornada ouviram minhas ideias/reclamações e contribuiram com sugestões/apoio.

Agradeço ao professor Antonio Carlos Sementille por aceitar orientar o presente trabalho e por me guiar durante todo esse período de desenvolvimento. Todos os seus conselhos e correções serviram de aprendizado e agregaram para o meu crescimento na vida acadêmica.

Agradeço também o meu parceiro de trabalho Marcelo Eduardo Benencase por me ajudar a definir o tema da atual pesquisa.

Por fim, gostaria de agradecer ao usuário `zCybeRz` do site Reddit, que me auxiliou muito na elaboração dos algoritmos paralelizados para a placa gráfica.

The best way to predict the future is to implement it.
David Heinemeier Hansson

Resumo

Consoante com o desenvolvimento tecnológico atual que, cada vez mais, solicita abordagens que conectem o meio analógico e digital de maneira interativa, ou seja, funcionando em tempo real, o presente trabalho busca auxiliar nesse cenário ao acelerar uma das técnicas de extração de pontos de interesse em imagem presente no estado da arte da categoria, uma vez que são densamente utilizadas em áreas como realidade aumentada, veículos autônomos, robôs de serviço, reconstrução 3D, e diversas outras que necessitam produzir resultados rápidos e frequentes. O método escolhido para o aperfeiçoamento é o extrator de cantos FOAGDD (*First-order Anisotropic Gaussian Direction Derivative*) e seu processo de otimização se sustentou na massiva paralelização possibilitada pela arquitetura CUDA (*Compute Unified Device Architecture*) da NVIDIA. Os resultados obtidos com a melhoria proposta se mostraram promissores. Primeiramente, a saída produzida pela implementação original do método e a paralelizada em CUDA se mostraram muito similares, após serem testadas e comparadas em um conjunto de 28 imagens. Por último, o código proposto trouxe um *Speed-up* no tempo de execução de aproximadamente 3190 (66,03 segundos $\rightarrow$ 20,70 milisegundos) em relação à implementação original do FOAGDD, utilizando como base uma imagem padronizada de resolução 512×512 pixels.

Palavras-chave: Computação de alto desempenho, NVIDIA CUDA, Extrator de pontos de interesse.

Abstract

Cooperating with the current technological development that, more and more, requires approaches that connect the analog and digital medium in an interactive manner, that is, operating in real-time, the current work attempts to assist in this scenario by accelerating one of the interest point extraction techniques present in the state of the art of the category, since they are densely used in areas such as augmented reality, autonomous vehicles, service robots, 3D reconstructions, and several others that need to produce quick and frequent results. The method chosen for the improvement is the corner extractor FOAGDD (First-order Anisotropic Gaussian Direction Derivative), and its optimization process relied on the massive parallelization made possible by NVIDIA's CUDA (Compute Unified Device Architecture) architecture. The results obtained with the proposed modifications have shown to be promising. First, the output produced by the original implementation of the method and the one parallelized in CUDA turned out to be very similar after being tested and compared in a set of 28 images. Finally, the proposed code brought a speedup in the runtime of approximately 3190 (66,03 seconds $\rightarrow$ 20,70 milliseconds) over the original implementation of FOAGDD, using as base a standardized image with a resolution of 512 $\times$ 512 pixels.

Keywords: High performance computing, NVIDIA CUDA, Interest point detection.

Lista de figuras

Lista de tabelas

Lista de abreviaturas e siglas

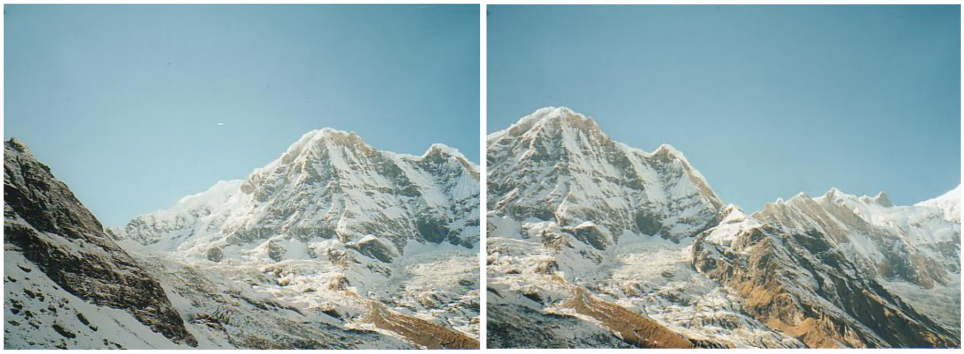
CJD	*Contrario Junction Detection*
ANDD	*Anisotropic Directional Derivative*
CPDA	*Chord-to-Point Distance Accumulation*
CPU	Unidade Central de Processamento
CUDA	*Compute Unified Device Architecture*
DoG	*Difference-of-Gaussian*
FAST	*Features from Accelerated Segment Test*
FOAGDD	*First-order Anisotropic Gaussian Direction Derivative*
FPGA	*Field Programmable Gate Arrays*
FPS	Frames Por Segundo
GCM	*Gradient Correlation Matrices*
GPGPU	*General Purpose Graphical Processing Unit*
GPU	*Graphical Processing Unit*
HD	*High Definition*
LIFT	*Learned invariant feature transform*
RA	Realidade Aumentada
RAM	*Random Access Memory*
SDK	*Software Development Kit*
SIFT	*Scale-Invariant Feature Transform*
SOGGDD	*Second-order Generalized Gaussian Directional Derivative*;
SURF	*Speeded Up Robust Features*
USF	*University of South Florida*

Sumário

1 Introdução

De acordo com Szeliski (2022), detecção e associação de características em imagens são componentes fundamentais de diversas aplicações em visão computacional. Por exemplo, considerando o par de imagens mostrado na Figura 1, pode-se querer alinhá-las de maneira que se juntem perfeitamente e componham um mosaico (Figura 2). Com o par da Figura 3, pode-se querer estabelecer um conjunto denso de correspondências para gerar uma reconstrução 3D da cena, que pode ser observada na Figura 4. Em ambos os casos, é necessário encontrar e associar características comuns entre as imagens para estabelecer esse alinhamento ou conjunto de correspondências.

Figura 1 – Par de imagens para o mosaico.



Fonte: Brown e Lowe (2007).

Figura 2 – Mosaico de imagem.

Fonte: Brown e Lowe (2007).

Ainda com Szeliski (2022), uma obordagem comum é representar essas características com pontos da imagem que são facilmente associados entre um par ou conjunto de imagens.

Figura 3 – Par de imagens para a reconstrução 3D.

Fonte: Bao et al. (2020).

Figura 4 – Reconstrução 3D.

Fonte: Elaborada pelo autor

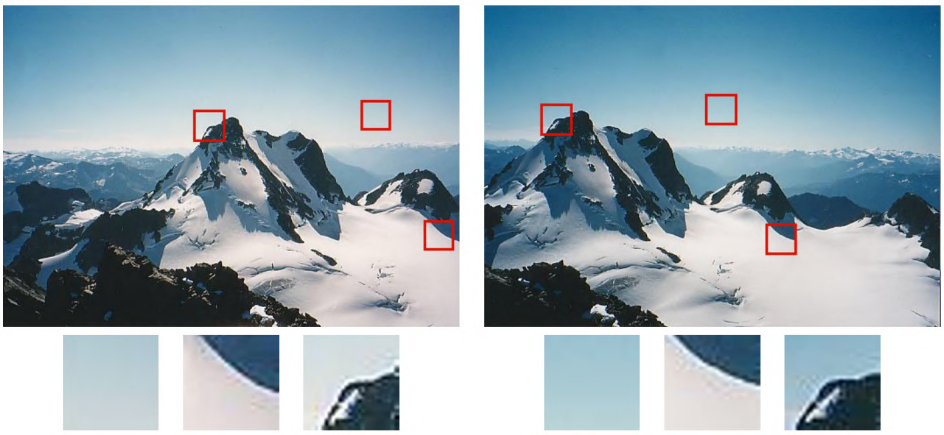
Esses pontos são coletados de regiões locais da figura, onde há informação suficiente para elas se destacarem. Como as aparências locais são representadas por blocos de pixels da imagem, os pontos de interesse são coletados de blocos com grandes alterações de intensidade (gradiente) nos seus pixels, pois possuem mais informação e, portanto, se destacam, ao contrário de blocos com pouca informação de textura. Na Figura 5 pode-se entender como o gradiente varia de acordo com a direção X e Y na imagem, já na Figura 6 é possível observar a importância de aparências locais com informações relevantes para o processo de associação.

De acordo com Burger e Burge (2010), blocos de pixels com variações no gradiente caracterizam regiões de borda ou canto na imagem. Regiões de borda são reconhecidas por terem um grande gradiente em uma direção e um gradiente pequeno na direção perpendicular à essa. Já pontos de cantos exibem grandes gradientes em múltiplas direções ao mesmo tempo. Ochoa (2016) dialoga que utilizar cantos ao invés de pontos de borda como representantes das características da imagem é preferível, pois eles são mais fáceis de associar. Isso é evidenciado

Figura 5 – Variação do gradiente de acordo com a direção da imagem. Esquerda, imagem de intensidade de um gato. No centro, gradiente da imagem na direção X, medindo a mudança da intensidade na horizontal. Na direita, gradiente da imagem na direção Y, medindo a mudança da intensidade na vertical. Pixels cinza representam gradientes pequenos; pixels preto e branco evidenciam gradientes grandes.

Fonte: Wikipedia contributors (2022).

Figura 6 – Aparências locais associadas entre duas imagens. Note como blocos com maior gradiente são associados mais facilmente que blocos com pouca textura – menor gradiente.

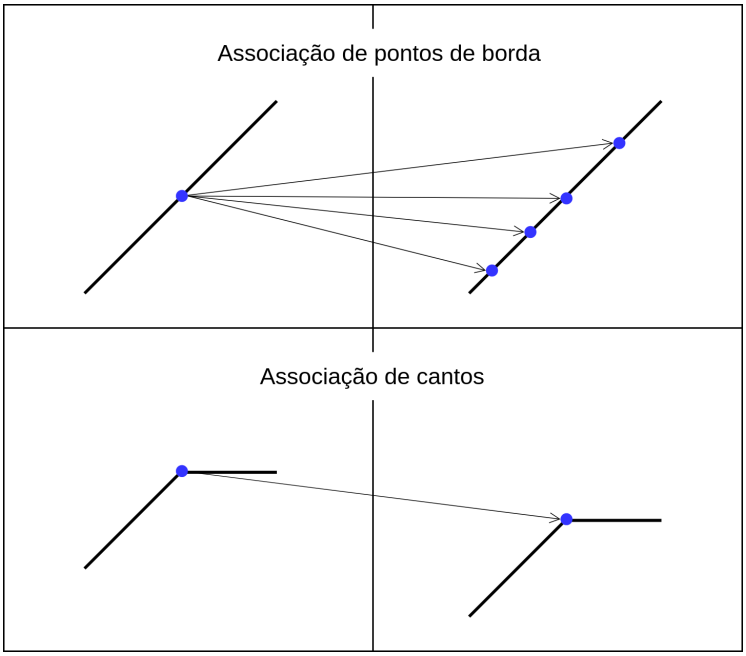


Fonte: Szeliski (2022).

na Figura 7. Por fim, um exemplo de detecção e associação de pontos de interesse entre duas imagens similares pode ser observado na Figura 8 – extração de pontos feita utilizando o algoritmo SIFT (*Scale-Invariant Feature Transform*) de Lowe (2004).

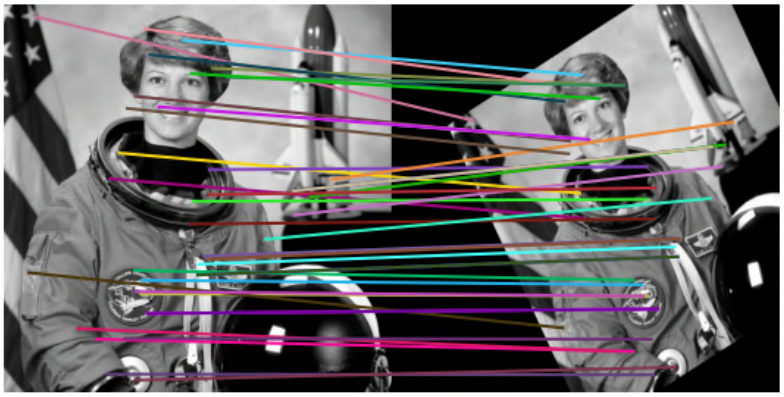
Em congruência com Forsyth e Ponce (2002), pontos de interesse são empregados no corpo de múltiplas tecnologias atuais, como formação de mosaico, calibração de câmera, reconstrução 3D, criação de registros tridimensionais para RA (Realidade Aumentada) e veículos autônomos, entre outras. Segundo Klippenstein e Zhang (2007) e DeTone, Malisiewicz e Rabinovich (2018), das áreas mencionadas, aplicações de reconstrução 3D, veículos autônomos e RA, possuem o seu bom funcionamento intrinsicamente ligado à qualidade do algoritmo de extração de pontos de interesse integrado em seus módulos.

Figura 7 – Representação da associação de pontos de borda e pontos de canto entre pares de imagens. Note como a associção de pontos de borda é mais propensa a erros do que utilizando cantos.



Fonte: Ochoa (2016).

Figura 8 – Extração e associação de pontos de interesse com o extrator SIFT em um par de imagens



Fonte: Walt et al. (2014).

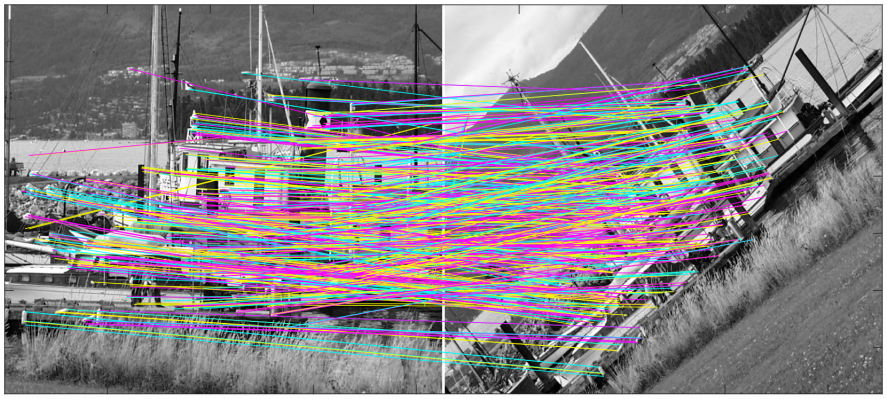
De acordo com Rey-Otero, Delbracio e Morel (2015), espera-se que um bom extrator encontre pontos bem distribuídos por toda imagem, para que informações sejam coletadas de diversas regiões. Além disso, os pontos encontrados precisam ser repetíveis sob variações na imagem, ou seja, caso a imagem original sofra alguma alteração, mas o contexto da cena continua similar, o extrator deve ser capaz de detectar os mesmos pontos. Para isso, os detectores devem ser invariantes à:

- Iluminação (Figura 9)
- Transformações Afins (Figura 10)
  - Translação
  - Rotação
  - Escala
  - Espelhamento
  - Cisalhamento
- Mudança de perspectiva/homografia (Figura 11)

Figura 9 – Exemplo do impacto da iluminação na detecção e associação de pontos de interesse. Note como alguns pontos entre as duas imagens não se repetem.

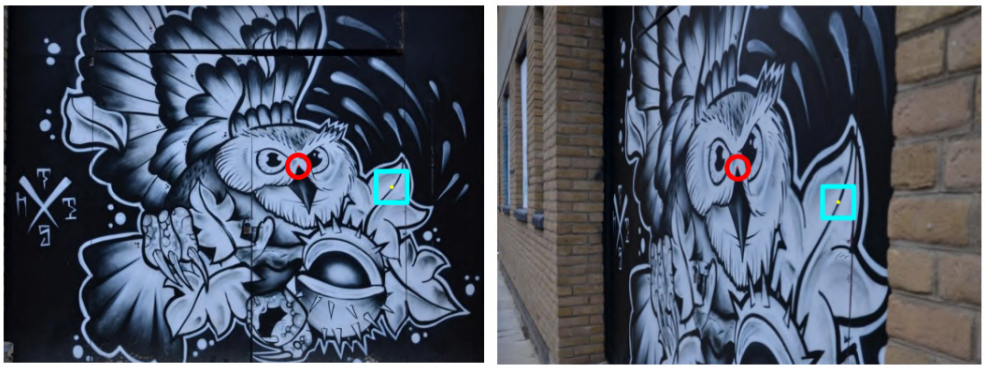
Fonte: Jing et al. (2022).

Figura 10 – Exemplo de detecção e associação de pontos invariantes à rotação.



Fonte: Jing et al. (2022).

Figura 11 – Exemplo de detecção e associação de pontos invariantes à mudança de perspectiva.



Fonte: Jing et al. (2022).

Um algoritmo recente de detecção de cantos em imagens é o FOAGDD (*First-order Anisotropic Gaussian Direction Derivative*) proposto por Zhang e Sun (2020). De acordo com os autores, a técnica desenvolvida é capaz de retratar corretamente a diferença entre cantos e bordas pela imagem, isso permite que o extrator não confunda essas duas classes de pontos na hora de fazer as detecções. No *benchmark*[1] qualitativo dos autores, o algoritmo proposto superou dez detectores no estado da arte da categoria, para isso o FOAGDD se destacou na repetibilidade das extrações em pares de imagens sob transformações afins, mudança de perspectiva, compressão e ruído.

Além da dependência de extrações de ótima qualidade, Quandt et al. (2018), Yaqoob et al. (2020), Medioni et al. (2007), Sveleba et al. (2019) e Ulrich, Steger e Baumgartner (2003) afirmam que alguns *softwares* necessitam ainda que os extratores funcionem em tempo real[2], pois possuem interações constantes com o usuário e/ou precisam produzir informações imediatas referentes ao ambiente em que estão inseridas. Por consequência desse dinamismo, pesquisas na área de extração de características têm sido direcionadas para encontrar soluções que diminuam ainda mais o tempo de suas computações, de maneira que aplicações que fazem uso desses detectores não sejam afetadas por variações de desempenho no dispositivo que estão inseridas.

1.1 Problema

Tang et al. (2019) afirmam que os extratores de características mais clássicos e comuns como SURF (*Speeded Up Robust Features*) de Bay, Tuytelaars e Gool (2006), SIFT (*Scale-*

[1] Em concordância com Lange e Kidd (2021), *benchmark* nesse contexto é a maneira formal de comparar as práticas, processos e resultados da pesquisa desenvolvida com outras da área, para avaliar se a performance obtida é abaixo ou acima da média. É principalmente orientado por dados.

[2] Krings (2022) dialoga que a frequência de 24 FPS (Frames Por Segundo) , é o padrão de filmagem cinemático para filmes, séries de TV e vídeos online. Já 30 FPS é direcionado para filmagens mais fluídas, como programas televisivos, esportes, concertos, entre outros. Logo, a frequência de tempo real no âmbito digital é compreendido entre 24 e 30 FPS.

Invariant Feature Transform) de Lowe (2004) e FAST (*Features from Accelerated Segment Test*) de Rosten e Drummond (2005) são algoritmos caros computacionalmente. Por consequência, tecnologias atuais que requerem a implementação de um detector com boas extrações e que funcione em tempo real, se encontram majoritariamente de mãos atadas.

Como mencionado previamente, o método de detecção de cantos em imagem FOAGDD é um algoritmo recente e presente no estado da arte da categoria por possuir extrações de ótima qualidade, como por exemplo, os pontos detectados são repetíveis sob transformações afins, mudança na perspectiva, presença de ruído, entre outros. No entanto, uma particularidade negativa da técnica é sua alta latência, isso se deve à execução serial de múltiplas computações densas pela imagem para extrair suas características, tomando muito tempo no processamento. Sendo assim, de acordo com as considerações finais dos autores, a implementação original do FOAGDD não é apropriada para aplicações que funcionam em tempo real, no entanto, há espaço para melhorias.

1.2 Justificativa

Tendo em vista o crescente desenvolvimento de tecnologias em tempo real que utilizam extratores de pontos de interesse como base para seu funcionamento e o poder computacional presente nas GPGPUs (*General Purpose Graphical Processing Unit*), o atual trabalho se justifica ao utilizar a massiva arquitetura de paralelização contida em uma GPGPU para otimizar a velocidade do extrator de pontos FOAGDD compreendido no estado da arte da categoria.

1.3 Objetivos

O objetivo principal deste trabalho foi implementar e testar uma versão paralela do algoritmo de extração de interesse FOAGDD, visando acelerar o desempenho do mesmo em relação à versão original de seus autores Zhang e Sun (2020). Para isso, fez-se uso de dispositivos com arquitetura de massiva paralelização – placas gráficas, em específico com a arquitetura CUDA, da NVIDIA.

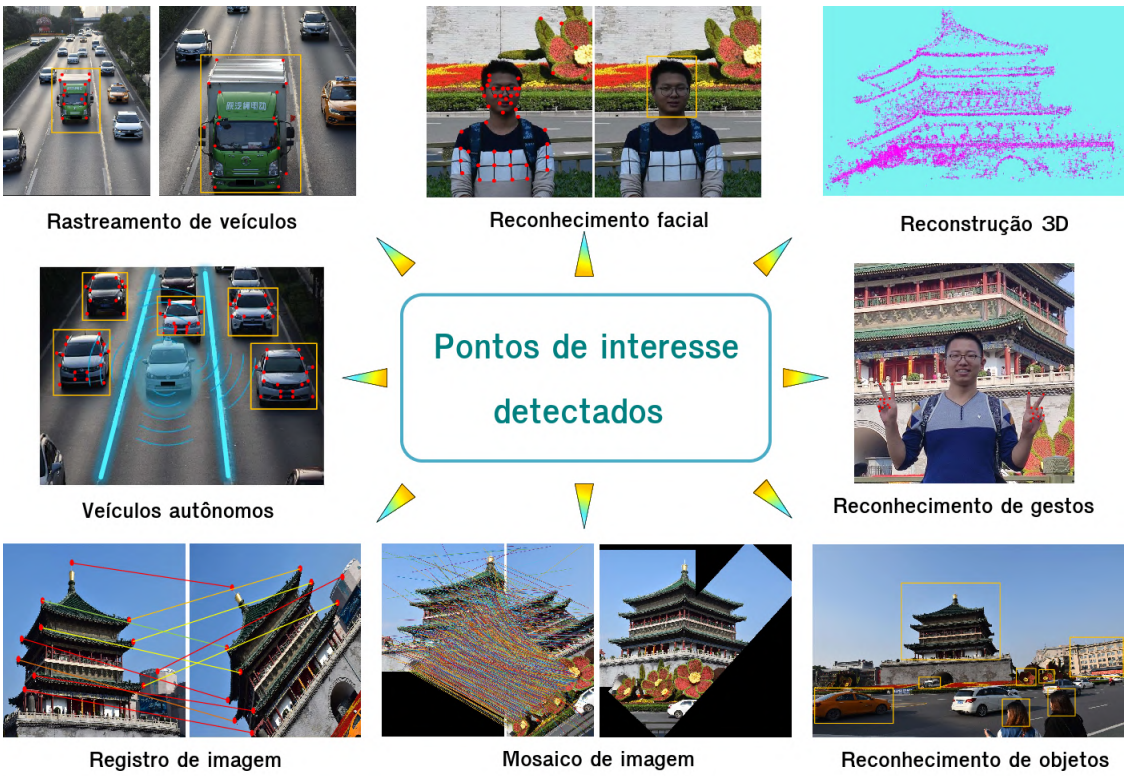
2 Fundamentação Teórica

Nesse capítulo, serão detalhados os assuntos que giram em torno do atual trabalho. A idéia é definir uma base para que se entenda o escopo da pesquisa e a motivação por trás dos passos tomados no desenvolvimento do método.

2.1 Algoritmos de extração de características

Pontos de interesse são considerados muito importantes para a percepção visual humana de formas e vem sendo uma das características de imagem chaves utilizada como pistas críticas para várias tarefas de processamento e entendimento de imagem, tais como: calibração de câmera, registro de imagem, reconhecimento de objeto, reconstrução 3D, veículos autônomos, reconhecimento facial, RA, entre outros exemplificados na Figura 12. (JING et al., 2022)

Figura 12 – Exemplos de aplicações que utilizam pontos de interesse



Fonte: Jing et al. (2022).

De acordo com Tyagi (2019), pontos de interesse devem obedecer as seguintes propriedades:

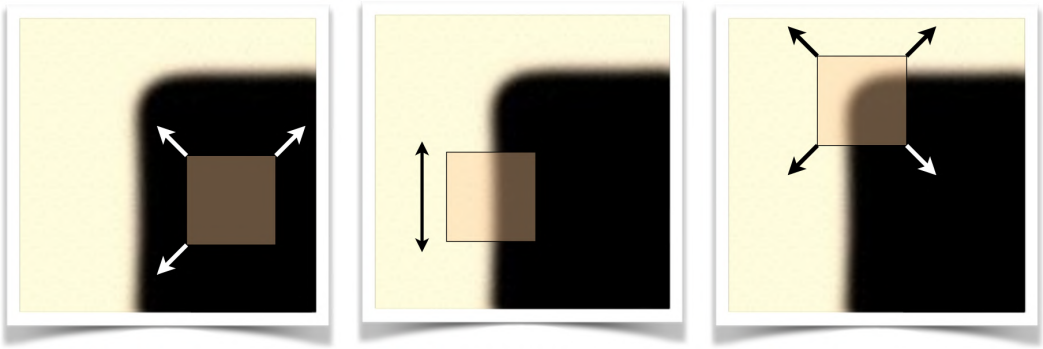
- Possuem uma posição bem definida no domínio da imagem.

- São estáveis sob perturbações locais e globais no domínio da imagem, como variações na iluminação/brilho, de tal forma que os pontos consigam ser computados com um alto grau de repetibilidade.
- Devem ser detectados rapidamente.

Exemplificando um algoritmo de extração de pontos de interesse, segundo Sánchez, Monzón e Nuez (2018), o detector Harris é uma técnica padrão utilizada para localizar cantos em uma imagem. Apesar da aparição de diversos detectores na última década, ele continua sendo uma técnica de referência.

Segundo Sánchez, Monzón e Nuez (2018) e Derpanis (2004), a ideia por trás do Harris é de detectar pontos na imagem baseado na variação de intensidade em uma vizinhança local: uma região pequena ao redor de uma característica deve apresentar uma grande mudança na intensidade quando comparada com janelas de imagem deslocadas em qualquer direção. A classificação dessas regiões de acordo com a variação de intensidade pode ser vista na Figura 13.

Figura 13 – Classificação da região de acordo com sua variação de intensidade



Região plana: nenhuma variação em todas as direções

Borda: nenhuma variação ao longo da direção da borda

Canto: varição significante em todas as direções

Fonte: Kitani (2017).

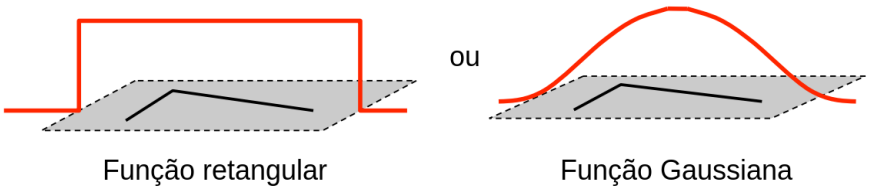
Sendo assim, considerando uma imagem I como uma função escalar $I : \Omega \to \mathbb{R}$ e $(\triangle x, \triangle y)$ como uma pequena variação em qualquer direção no domínio, $(x, y) \in \Omega$; cantos são pontos (x, y) que maximizam a Equação 2.1 para pequenas variações de $(\triangle x, \triangle y)$.

$$E(\triangle x, \triangle y) = \sum_{x,y} w(x, y)(I(x + \triangle x, y + \triangle y) - I(x, y))^2 \tag{2.1}$$

A função de janela $w(x, y)$ ajuda a selecionar a região de suporte e geralmente é uma função retangular ou Gaussiana, como observado na Figura 14.

Utilizando expansões de Taylor, pode-se linearizar a função $I(x + \triangle x, y + \triangle y)$ se tornando $I(x+\triangle x, y+\triangle y) \simeq I(x, y)+I_x(x, y)\triangle x+I_y(x, y)\triangle y$, onde I_x e I_y são derivadas

Figura 14 – Funções de janela.



Fonte: Kitani (2017).

da imagem com respeito aos eixos x e y. Substituindo na Equação 2.1 tem-se:

$$E(\triangle x, \triangle y) \simeq \sum_{x,y} w(x)(I_x(x,y) \triangle x + I_y(x,y) \triangle y)^2 = [\triangle x, \triangle y] M [\triangle x, \triangle y]^T \quad (2.2)$$

Essa última expressão depende do gradiente da imagem a partir de um tensor de estrutura[1], que é dada pela Equação 2.3.

$$M = \sum_{x,y} w(x)([I_x, I_y]^T [I_x, I_y]) = \sum_{x,y} w(x) \begin{bmatrix} I_x^2 & I_x I_y \\ I_x I_y & I_y^2 \end{bmatrix} \quad (2.3)$$

Os pontos de máximo da Equação 2.2 são encontrados através da análise dessa matriz. O maior autovalor[2] de M corresponde a direção de maior variação de intensidade, enquanto o segundo corresponde a variação de intensidade na sua direção ortogonal. Analisando seus valores, denominados λ_1 e λ_2, pode-se cair em três possíveis situações:

- $\lambda_1 \approx \lambda_2 \approx 0$, caracterizando uma região homogênea com variações de intensidade devido a ruídos.
- $\lambda_1 \gg \lambda_2 \approx 0$, tipificando uma região de borda.
- $\lambda_1 > \lambda_2 \gg 0$, identificando uma região de canto.

As classificações dos pontos de acordo com os autovalores encontrados pode ser melhor visualizada na Figura 15.

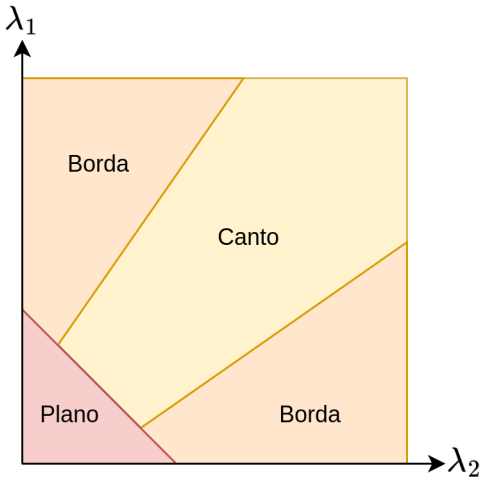
2.2 Survey de extratores

Vendo a necessidade de um estudo completo sobre as técnicas de extração de pontos em imagem, que são integradas em variadas tecnologias atuais, Jing et al. (2022) fizeram um

[1] Segundo Arseneau (2008), o tensor de estrutura representa a distribuição do gradiente na região ao redor de um pixel.

[2] Segundo Hefferon (2018), uma transformação linear $A : V \to V$ possui um autovalor escalar λ se existir um autovetor $\overrightarrow{v} \in V$ tal que $A\overrightarrow{v} = \lambda\overrightarrow{v}$. Sendo assim, quando a transformação A é aplicada, $\overrightarrow{v}$ sofre uma mudança de escala λ, mas não muda sua direção.

Figura 15 – Relação dos autovalores com a classificação do ponto detectado



Fonte: Elaborada pelo autor

levantamento e compararam diversos detectores de pontos de interesse presentes no estado da arte da categoria. Nesse *survey*, foram definidos grupos de extratores de acordo com a técnica que utilizam:

- Baseada em intensidade
  - Baseada em derivada de primeira ordem;
  - Baseada em derivada de segunda ordem;
  - Baseada em derivada de alta ordem;
  - Baseada em auto-dissimilaridade em pixels da imagem.
- Baseada em contorno de cantos
  - Baseada em forma de borda;
  - Baseada em variação de intensidade em pixels de borda.
- Baseada em aprendizagem de máquina
  - Baseada em derivada de primeira ordem;
  - Baseada em combinação de derivada de primeira e segunda ordem;
  - Baseada na análise de características de pixels da imagem;
  - Baseada na representação de características de alto nível.
- Baseada em domínio de frequência

Nesses grupos, foram encaixados 18 detectores do estado da arte para serem avaliados, FAST de Rosten e Drummond (2005), Harris de Harris, Stephens et al. (1988), CJD (*Contrario*

Junction Detection) de Xia, Delon e Gousseau (2014), Harris-Laplace de Mikolajczyk e Schmid (2004), DoG (*Difference-of-Gaussian*) de Lowe (2004), SURF de Bay, Tuytelaars e Gool (2006), KAZE de Alcantarilla, Bartoli e Davison (2012), FOAGDD de Zhang e Sun (2020), SOGGDD (*Second-order Generalized Gaussian Directional Derivative*) de Zhang e Sun (2019), CPDA (*Chord-to-Point Distance Accumulation*) de Awrangjeb e Lu (2008), ANDD (*Anisotropic Directional Derivative*) de Shui e Zhang (2013), GCM (*Gradient Correlation Matrices*) de Zhang et al. (2010), New-Curvature de Zhang et al. (2019), LIFT (*Learned invariant feature transform*) de Yi et al. (2016), D2-Net de Dusmanu et al. (2019), LF-Net de Ono et al. (2018), He & Yung de He e Yung (2008), and DT-CovDet de Zhang et al. (2017)

Como critérios de avaliação para os métodos acima, foram escolhidas as seguintes métricas:

- Métricas de capacidade de detecção e acurácia de localização utilizando imagens de teste com *ground truths*. No geral, essa combinação de métricas é utilizada para avaliar detectores de canto. Logo, algumas imagens de teste são separadas e seus cantos são rotulados. Por fim, os extratores são passados pela imagem e os pontos detectados são comparados com os rótulos feitos.
- Avaliação de performance em aplicações visuais. Esse processo de avaliação é feito aplicando o detector de pontos de interesse em tarefas de visão computacional, como rastreamento e combinação de características, e comparando os resultados obtidos com dados rotulados.
- Métrica de repetibilidade em imagens alteradas com transformações afim. Como algumas aplicações que utilizam pontos de interesse requerem repetibilidade das detecções em imagens com relações temporais próximas (e.g., quadros sequenciais em um vídeo), existem datasets próprios para testar essa propriedade dos extratores.
- Tempo de execução e uso de memória

Com o escopo do levantamento bem definido, os autores do *survey* realizaram uma bateria de testes passando os detectores escolhidos sob os mesmos conjuntos de dados, cada qual com propriedades específicas para avaliar as métricas estipuladas. Dentre os extratores, o algoritmo FOAGDD foi o que mais se destacou nas três primeiras métricas, relacionadas à qualidade das detecções; no entanto, apresentou a pior performance na categoria de tempo de execução, chegando a demorar aproximadamente 40 segundos para extrair pontos de uma imagem (1200 x 1600). Todos os detectores mencionados no *survey* foram escritos em MATLAB e a máquina utilizada para os testes possui uma CPU 2.20 GHz, 128 GB de memória RAM e uma placa de vídeo RTX1080Ti com 12 GB de VRAM disponível.

2.3 FOAGDD

Em suma, para propor o algoritmo FOAGDD, Zhang e Sun (2020) analisaram algumas propriedades a respeito das técnicas utilizadas na detecção de cantos em uma imagem:

- **Propriedade 1:** A intensidade de variação de um canto é grande na maioria das direções, não necessariamente em todas as direções.
- **Propriedade 2:** Derivadas de primeira ordem ao longo das direções verticais e horizontais não conseguem captar bem as variações de intensidade de bordas e cantos.
- **Propriedade 3:** O filtro gaussiano isotrópico não consegue captar com acurácia a diferença de variação de intensidade entre bordas e cantos.
- **Propriedade 4:** O filtro anisotrópico gaussiano tem a habilidade de captar a diferença de variação de intensidade entre bordas e cantos.
- **Propriedade 5:** As técnicas existentes baseadas em tensor de estrutura retangular 2 x 2 não conseguem captar com acurácia as diferenças entre bordas e cantos.

Com isso em mente, Zhang e Sun (2020) propuseram o seguinte método para extrair cantos de uma imagem:

1. Use os filtros anisotrópico gaussiano multidirecional com multiescalas para suavizar a imagem de entrada e encontrar as derivadas direcionais multidirecionais em multiescalas, como evidenciado na Equação 2.7 que é desenvolvida a seguir:

   a) No domínio espacial, a definição do filtro anisotrópico Gaussiano pode ser vista na Equação 2.4.

   $$g_{\sigma,\rho,\theta}(x,y) = \frac{1}{2\pi\sigma^2}\exp(-\frac{1}{2\sigma^2}[x,y]R_{-\theta}\begin{bmatrix}\rho^{-2} & 0\\ 0 & \rho^2\end{bmatrix}R_\theta[x,y]^T) \tag{2.4}$$

   com

   $$R_\theta = \begin{bmatrix}\cos\theta & \sin\theta\\ -\sin\theta & \cos\theta\end{bmatrix} \tag{2.5}$$

   onde $[\ ,\ ]^T$ representa a matriz transposta, ρ é o fator anisotrópico ($\rho > 1$), σ a variação da função Gaussiana e R_θ(Equação 2.5) é a matriz de rotação com ângulo θ.

   b) A equação da derivada direcional de um filtro anisotrópico Gaussiano (Equação 2.6) é mostrada a seguir:

$$\phi_{\sigma,\rho,\theta} = \frac{\partial g_{\sigma,\rho}}{\partial y} R_\theta [x, y]^T \tag{2.6}$$

c) Por fim, a derivada anisotrópica Gaussiana direcional da imagem de entrada $I(x, y)$ é obtida utilizando o operador de convolução $\otimes$(Equação 2.7):

$$\bigtriangledown_{\sigma,\rho,\theta} I(x, y) = I(x, y) \otimes \phi_{\sigma,\rho,\theta}(x, y) \tag{2.7}$$

2. Para cada pixel da imagem, construa o tensor de estrutura multidirecional em multiescalas mostrado na Equação 2.8.

$$M = \begin{bmatrix} \sum_{i=-\frac{u}{2}}^{\frac{u}{2}} \sum_{j=-\frac{v}{2}}^{\frac{v}{2}} \bigtriangledown^2_{\sigma_s,\rho,1} I(n_x + i, n_y + j) & \cdots & \sum_{i=-\frac{u}{2}}^{\frac{u}{2}} \sum_{j=-\frac{v}{2}}^{\frac{v}{2}} \bigtriangledown_{\sigma_s,\rho,1} I(n_x + i, n_y + j) \bigtriangledown_{\sigma_s,\rho,k} I(n_x + i, n_y + j) \\ \vdots & \ddots & \vdots \\ \sum_{i=-\frac{u}{2}}^{\frac{u}{2}} \sum_{j=-\frac{v}{2}}^{\frac{v}{2}} \bigtriangledown_{\sigma_s,\rho,k} I(n_x + i, n_y + j) \bigtriangledown_{\sigma_s,\rho,1} I(n_x + i, n_y + j) & \cdots & \sum_{i=-\frac{u}{2}}^{\frac{u}{2}} \sum_{j=-\frac{v}{2}}^{\frac{v}{2}} \bigtriangledown^2_{\sigma_s,\rho,k} I(n_x + i, n_y + j) \end{bmatrix} \tag{2.8}$$

onde $n = [n_x, n_y]^T$ representa a coordenada do pixel no espaço euclidiano dos inteiros $\mathbb{Z}^2$.

3. Obtenha os autovalores para cada escala.

4. Marque um pixel p como candidato à canto se sua medida de canto (Equação 2.9) é o máximo local em uma janela centralizada em p e é maior que um dado limite T_h na menor escala.

$$\mathfrak{p}_s(n_x, n_y) = \frac{\prod_{k=1}^{k} \lambda_k}{\sum_{k=1}^{k} \lambda_k + \tau} \tag{2.9}$$

onde $K = \lambda_1, \lambda_2, \cdots, \lambda_k$ é o conjunto de autovalores do tensor de estrutura multidirecional $K \times K$ e τ é uma constante pequena ($\tau = 2.22 \times 10^{-16}$) para evitar divisões por zero.

5. Marque um pixel p como candidato à canto se sua medida de canto é maior que um dado limite T_h em todas as escalas.

Com o método definido, Zhang e Sun (2020) seguiram com testes quantitativos e qualitativos para medir a repetibilidade e acurácia das detecções, assim como o uso de memória e tempo do algoritmo, em relação a outros extratores no estado da arte da categoria. Como resultado, o FOAGDD conseguir se sobressair qualitativamente dentre os detectores a qual foi comparado. No entanto, é apontado que o resultado do tempo utilizado para suas computações é excessivamente alto para aplicações que funcionam em tempo real. O código desenvolvido

para esses testes foi escrito em MATLAB e executado em uma máquina com uma CPU 2.81 GHz e 16 GB de memória RAM disponível. Por fim, os autores sugeriram, como trabalhos futuros, que o método proposto pode ser otimizado ao fazer uso de computações em GPU ou FPGA (*Field Programmable Gate Array*) para acelerar sua execução.

2.4 CUDA

De acordo com Ghorpade et al. (2012), a computação na GPU (*Graphical Processing Unit*) forneceu uma vantagem sobre a CPU (*Central Processing Unit*), com respeito a velocidade de seus cálculos. Sendo assim, é uma das áreas de estudo mais interessantes na área de pesquisa e desenvolvimento da indústria moderna.

Segundo Ghorpade et al. (2012), a GPU é uma unidade gráfica de processamento que permite executar gráficos de alta definição em um computador, requisito importante para a computação moderna. Como a CPU, ela também possui um chip único de processamento. No entanto, a arquitetura de uma placa gráfico inclui milhares de núcleos de processo se comparado aos 32 ou 64 núcleos presentes nas últimas CPUs, como observado na Figura 16. O trabalho primário da GPU é o cálculo de funções 3D. Porque esse tipo de computação é muito pesada para a CPU, esse hardware consegue ajudar o computador a funcionar com maior eficiência. Apesar da GPU ter sido criada inicialmente para propósitos gráficos, atualmente expandiu para computações de propósito geral.

Figura 16 – Comparação dos núcleos presentes em uma CPU e uma GPU. Note como a GPU possui mais unidades de processamento, ALU (*Arithmetic Logic Unit*), em detrimento de áreas destinadas a controle e cache, ao passo que a CPU funciona de maneira inversa.

Fonte: Storti e Yurtoglu (2015).

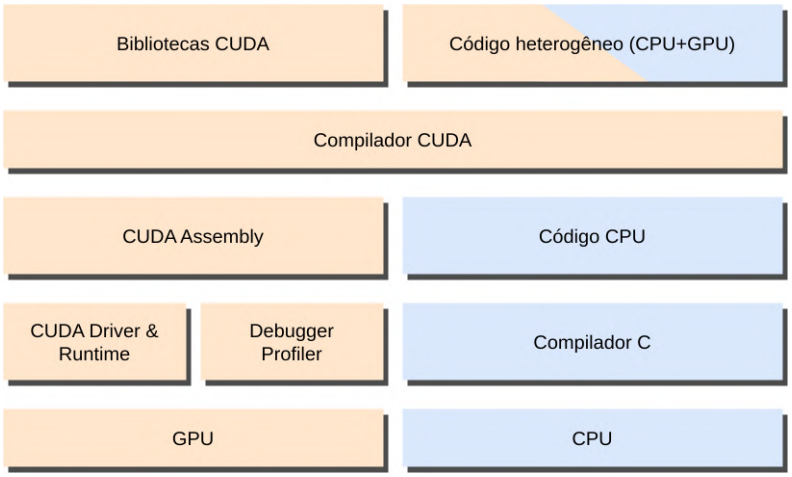
Congruente com Cheng, Grossman e McKercher (2014), o uso da placa gráfica para acelerar aplicações é combinando o processamento da CPU com a GPU em um modelo de computação de coprocessamento heterogêneo. Portanto, partes da aplicação com poucos dados, lógica de controle sofisticada e/ou pouco paralelismo são executada na CPU – referenciado

como código do *host*, enquanto partes com muitos dados e altamente paralelizáveis são aceleradas pela GPU – referenciado como código do *device*.

Análogo com Ghorpade et al. (2012) e Cheng, Grossman e McKercher (2014), em 2006-2007 a NVIDIA introduziu sua arquitetura massiva de paralelização chamada CUDA e mudou todo o cenário de programação na GPGPU. CUDA é uma plataforma de computação paralela de uso geral e modelo de programação que aproveita do mecanismo de computação paralela em GPUs da NVIDIA para resolver problemas computacionais complexos de maneira eficiente. A plataforma CUDA é acessível através de bibliotecas aceleradas com CUDA, diretivas do compilador, interfaces de programação de aplicações e extensões para linguagens de programação, incluido C, C++, Fortran e Python.

Ainda com Cheng, Grossman e McKercher (2014), o compilador CUDA da NVIDIA `nvcc` é o responsável por separar o código do *host* do código do *device* durante a etapa de compilação. Tomando como exemplo a programação em CUDA C, o código *host* é escrito em C puro e posteriormente compilado com um compilador da própria linguagem. Já o código *device* é escrito com uma extensão de C, com palavras-chave para rotular funções que irão trabalhar com o paralelismo dos dados, essas funções são chamadas/referidas como *kernels*. O código do *device* é futuramente compilado com o `nvcc`. A Figura 17 ilustra esse processo.

Figura 17 – Etapas de compilação de um código desenvolvido em CUDA C.

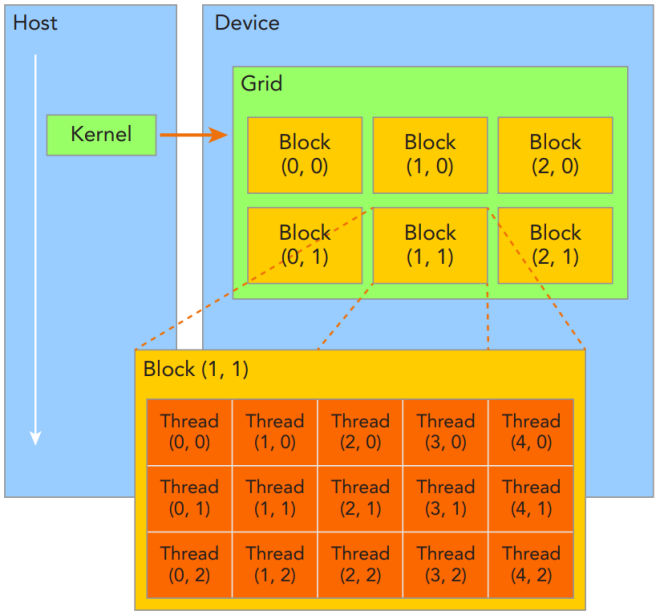


Fonte: Cheng, Grossman e McKercher (2014).

Storti e Yurtoglu (2015) descrevem que, após a chamda de um *kernel* no *host*, sua execução é passada para o *device* onde uma série de *threads* da GPU serão assignadas para executar as declarações feitas no *kernel*. Essas *threads* são decompostas em uma hierarquia de dois níveis, blocos de *threads* (*blocks of threads*) e grade de blocos (*grid of blocks*), como mostrado na Figura 18. Todas as *threads* utilizadas na execução de um kernel são coletivamente chamadas de grade e compartilham um mesmo espaço de memória global. Uma grade é formada por vários blocos de *threads*. Um bloco é composto de várias *threads* que cooperam entre si utilizando recursos de sincronização e memória compartilhada, ambos funcionando apenas

entre *threads* de um mesmo bloco, ou seja, *threads* em blocos diferentes não cooperam entre si.

Figura 18 – Organização das threads na arquitetura CUDA.



Fonte: Cheng, Grossman e McKercher (2014).

Gupta (2020) explica que as *threads* e blocos são organizados em até 3 dimensões para indexar elementos em vetor, matriz e volume. Na Figura 18, por exemplo, é utilizado uma hierarquia de grade 2D e blocos 2D. Cada dimensão de grade e bloco deve ser criada com a `struct` de CUDA `dim3` ou utilizando `int` caso seja unidimensional. Essas informações são então passadas para a função *kernel* em sua chamada, utilizando a sintaxe `<<<...>>>` de CUDA. Durante a execução do kernel, é possível acessar *index* do bloco e *thread* que estão sendo usados no momento através das palavras reservadas `blockIdx` e `threadIdx` respectivamente. Essas palavras são do tipo CUDA `uint3` e possuem os campos `x`, `y` e `z` para acessar o index das dimensões, como `threadIdx.x` ou `blockIdx.y`. A amostra de código na Figura 19 engloba os conceitos mencionados.

Como existem muitas palavras reservadas atreladas a recursos disponibilizados pela extensão CUDA em C, é importante se direcionar e entender as mais utilizadas durante o desenvolvimento dos *kernels*. A seguir, Storti e Yurtoglu (2015) lista e descreve algumas delas:

- `__global__`: é o qualificador para kernels que podem ser chamados tanto no *host* quanto no *device*.
- `__host__`: qualifica funções que são chamadas e executadas apenas no *host*.
- `__device__`: qualifica funções que são chamadas e executadas apenas no *device*.

Figura 19 – Amostra de código CUDA evidenciando o controle de index das threads.

```
__global__ void funcao_kernel(...)
{

    // Computacao do index da thread atual na dimensao X
    int thread_idx_x = blockIdx.x * blockDim.x + threadIdx.x;
    // Computacao do index da thread atual na dimensao Y
    int thread_idx_y = blockIdx.y * blockDim.y + threadIdx.y;
    ...
}

int main()
{
    ...
    // define o numero de threads contidas em um bloco
    dim3 threads_por_bloco(16, 16);
    // define o numero de blocos contidos na grade
    dim3 numero_blocos(8, 8);
    // chamada do kernel
    funcao_kernel<<<numero_blocos, threads_por_bloco>>>(...);
    ...
}
```

Fonte: Elaborada pelo autor.

- *Kernels* não podem retornar nenhum valor, portanto seu tipo de retorno é sempre `void`. Declarações de *kernel* seguem a seguinte definição:
  `__global__ void um_kernel<<<num_blocos,num_threads>>>(argumentos)`
- *Kernel* providenciam variáveis de tamanho e *index* para cada bloco e thread.
  Variáveis de tamanho
  - `gridDim`: especifica o número de blocos em uma grade.
  - `blockDim`: especifica o número de blocos em cada bloco.

  Variáveis de index
  - `blockIdx`: retorna o *index* do bloco na grade.
  - `threadIdx`: retorna o *index* da *thread* dentro do bloco.

A API do CUDA providencia algumas funções para transferir dados entre a CPU e GPU:

- `cudaMalloc`: aloca memória da GPU.
- `cudaMemcpy`: transfere dado entre a CPU e GPU.
- `cudaFree`: libera memória da GPU que não está sendo mais usada.

O CUDA também providencia algumas funções para controlar as execuções em paralelo:

Figura 20 – Computação das distâncias na CPU (esquerda) e GPU (direita).

```
#include <math.h>
#define N 64 // Tamanho de vetor

// Normaliza valores entre 0 e 1
float scale(int i, int n)
{
  return ((float)i)/(n - 1);
}

 // Computa a distancia entre dois
    pontos em uma linha
float distance(float x1, float x2)
{
  float diff = x2 - x1;
  return sqrt(diff*diff);
}
 int main()
{
  // Cria um vetor de N floats (
    inicializados com 0.0)
  float out[N] = {0.0f};
  // Valor de referencia para medir as
    distancias
  float ref = 0.5f;
  /* for loop para normalizar o index
    para obter o valor da coordenada,
  * computar a distancia em relacao ao
    ponto de referencia
  * e guardar o resultado no index
    apropriado do vetor out */
  for (int i = 0; i < N; ++i)
  {
    float x = scale(i, N);
    out[i] = distance(x, ref);
  }
  return 0;
}
```

```
#include <stdio.h>
#define N 64
#define TPB 32

__device__
float scale(int i, int n)
{
  return ((float)i)/(n - 1);
}

__device__
float distance(float x1, float x2)
{
  float diff = x2 - x1;
  return sqrt(diff*diff);
}

__global__
void distanceKernel(float *d_out,
                    float ref,
                    int len)
{
  int i = blockIdx.x*blockDim.x +
    threadIdx.x;
  float x = scale(i, len);
  d_out[i] = distance(x, ref);
}

int main()
{
  const float ref = 0.5f;
  // declara um ponteiro para guardar um
     vetor de floats
  float *d_out = 0;
  // Aloque memoria no device para
    guardar a saida do kernel
  cudaMalloc(&d_out, N*sizeof(float));
  // Execute o kernel para computar e
    guardar o valor das distancias
  distanceKernel<<<N/TPB, TPB>>>(d_out,
    ref, N);
  cudaFree(d_out); // Libere a memoria
  return 0;
}
```

Fonte: Storti e Yurtoglu (2015).

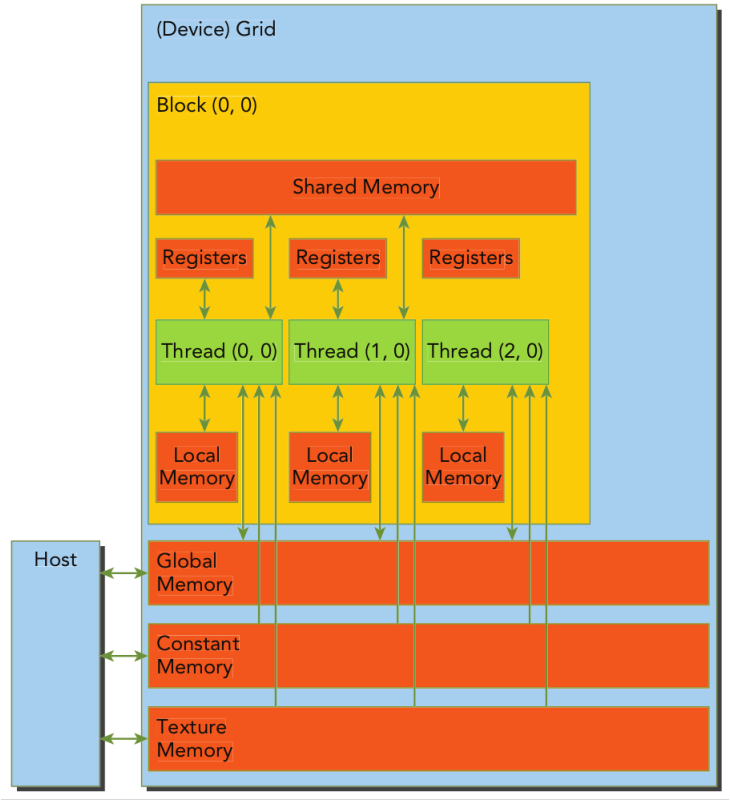
- `__syncthreads`: sincroniza *threads* de um bloco.
- `cudaDeviceSynchronize`: sincroniza todas as *threads* de uma grade.

Para exemplificar alguns dos conceitos acima, Storti e Yurtoglu (2015) desenvolveram uma amostra de código simples (Figura 20) que compara distância entre pontos de uma sequência com relação a um ponto de base escolhido. Essa algoritmo foi desenvolvido inicialmente para funcionar na CPU e utiliza laços de *for* para percorrer os pontos. Em seguida esse mesmo código é reestruturado para tirar proveito da paralelização da GPU, onde as iterações do laço são movidas para um *kernel*.

Segundo Cheng, Grossman e McKercher (2014), o CUDA possui uma hierarquia de

memória (Figura 21) exposta ao desenvolvedor para que ele possa organizar o acesso à memória seguindo as prioridades do algoritmo que está sendo elaborado. A hierarquia é estruturada de acordo com o princípio da localidade, ou seja, se um dado na memória é referenciado, localidades próximas possuem maior probabilidade de serem referenciadas também. Logo, a hierarquia de memória é progressiva: baixa-latência e pouco-armazenamento → alta-latência e muito-armazenamento. Dito isso, a seguir são brevemente descritas as memórias expostas ao programador:

Figura 21 – Hierarquia de memória em CUDA.



Fonte: Cheng, Grossman e McKercher (2014).

- **Registradores:** memória de acesso mais rápido na GPU, porém é um recurso escasso. Normalmente armazenam variáveis automáticas declaradas no *kernel*. Registradores são para uso exlusivos de cada *thread*.
- **Memória local:** armazena variáveis que são elegíveis aos registradores, mas não cabem na memória. Apesar do nome "Memória local", ela é localizada na mesma região física que armazena a memória global, portanto caracteriza alta latência de acesso.
- **Memória compartilhada:** variáveis com o prefixo `__shared__` são guardadas na memória compartilhada. Devido a estar contida no chip da GPU, essa memória possui alta taxa de transporte e baixa latência de acesso em relação à memória local e memória

global. Também serve como comunicação *inter-thread*, logo, *threads* em um mesmo bloco podem cooperar compartilhando dados armazenados na memória compartilhada.

- **Memória constante:** variáveis com o prefixo `__constant__` são guardadas na memória constante. Esse tipo de armazenamento reside na GPU em uma área de cache. A memória constante necessita ser declarada globalmente e é visível por todos os kernels. Os dados armazenados na memória constante são apenas para leitura.
- **Memória de textura:** é armazenada na GPU como um cache apenas de leitura. A memória de textura é útil quando se trabalha com dados 2D, pois possui suporte à funções de interpolação, acesso localizado 2D, entre outros.
- **Memória global:** memória de acesso mais lento na GPU, porém é um recurso abundante. A parte "global"do nome se refere à sua permanência na memória mesmo após o fim de execução do kernel. A alocação e liberação de memória com ponteiros para variáveis que utilizam a memória global é feita através das funcões de CUDA `cudaMalloc` e `cudaFree` respectivamente. Normalmente, as variáveis que utilizam essa memória são passadas para uma função *kernel* através de seus parâmetros e são visíveis por todas as *threads* na grade.

3 Desenvolvimento da Pesquisa

Nesse capítulo são descritas as técnicas utilizadas para o desenvolvimento da pesquisa.

3.1 Considerações iniciais

Seguindo a implementação dos autores, o código do FOAGDD pode ser dividido nos seguintes componentes que são executados serialmente:

- **Preparação da imagem:** caso a imagem de entrada possua três canais de cores, então ela é convertida de RGB $\rightarrow$ Cinza, para que seja levado em consideração apenas a intensidade dos pixels nas computações. Também, a imagem é extendida nas bordas para auxiliar nas operações seguintes.

- **Computação dos filtros:** são computados os filtros gaussianos anisotrópicos que serão utilizados para obter as derivadas direcionais multidirecionais em multiescalas da imagem de entrada. Para isso, é variada a escala do filtro M vezes e cada escala é então rotacionada em N ângulos uniformemente distribuídos em um circunferência; porfim, tem-se M escalas $*$ N rotações $= M * N$ filtros.

- **Computação das derivadas:** aplicando o operador de convolução $\otimes$ entre a imagem de entrada e cada um dos filtros gerados na etapa *Computação dos filtros*, são produzidas $M * N$ imagens representando cada derivada direcional escalada.

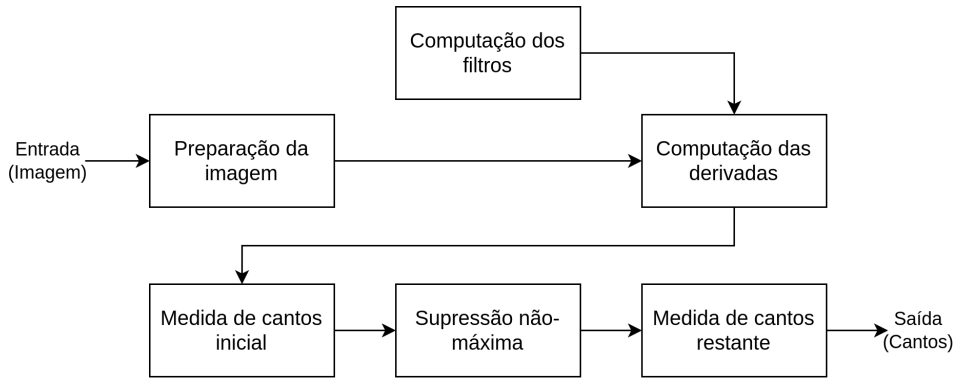
- **Medida de cantos inicial (primeira escala):** é passada uma janela de tamanho $T \times T$ em cada pixel p na primeira escala das derivadas direcionais; a janela é centralizada em p. Todos os pixels contido nessa janela ao redor de p são dispostos ao longo de uma coluna em uma matriz $N \times T^2$. Para preencher as demais colunas, o mesmo procedimento é realizado no mesmo pixel p nas demais derivadas direcionais na primeira escala. Em seguida, a matriz resultante é multiplicada por si mesmo, obtendo uma nova matriz $N \times N$ simétrica. Por fim, com essa nova matriz é computada a medida de canto para o pixel p como mostrado na Equação 2.9.

- **Supressão não-máxima:** com as medidas de canto, valores acima de um limite T_h são marcados como candidatos a canto, enquanto os demais são descartados. Como muitos pixels ao redor de um canto na imagem são candidatos a ponto de interesse, é necessário suprimir todos as medidas de canto menores que o máximo local daquela região, para que se tenha apenas um pixel por canto na imagem. Esse processo é feito através de uma supressão não-máxima.

- **Medida de cantos restante:** com os pontos candidatos a canto obtidos com a primeira escala das derivadas direcionais, repete-se o processo nas escalas restantes para obter as medidas de canto e marcar os candidatos a ponto de interesse. No entanto, agora não é necessário percorrer por todos os pixels das derivadas para realizar essas operações, utiliza-se os pixels candidatos a canto obtidos anteriormente como ponto de partida.

O diagrama apresentado na Figura 22 demonstra visualmente como as chamadas dos componentes são organizadas dentro do algoritmo.

Figura 22 – Estrutura de Chamadas do algoritmo FOAGDD.



Fonte: Elaborada pelo autor

Notou-se que os componentes com maiores contribuições para a latência do extrator são: *Computação das derivadas* e *Medida de cantos inicial*. A seguir são descritas as possíveis causas:

- **Computação das derivadas:** nessa etapa, como padrão, os autores definiram um total de 24 filtros a serem passados pela imagem de entrada para encontrar as derivadas direcionais multidirecionais em multiescalas que serão utilizadas nas etapas seguintes. O número total de filtros se dá pelas seguintes variações: 3 escalas $*$ 8 rotações $=$ 24 filtros. Com essa quantia de filtros, são produzidas 24 imagens representando cada derivada direcional escalada. Tendo em vista essa gama de operações de convolução, entende-se o motivo do tempo desse componente se sobressair em relação aos demais.
- **Medida de cantos inicial:** tomando como base a imagem de teste e os parâmetros do método sugerido pelos autores, nessa etapa é requisitado acesso à 7^2 elementos em janela $*$ (512 elementos $+$ 14 valores de preenchimento)^2 $*$ 8 rotações, equivalendo a cerca de 108 milhões de acessos á elementos. Com base nesse comportamento, fica claro o motivo do tempo exacerbado gasto nesse componente do método.

Visando esclarecer ainda mais o funcionamento desses dois módulos, o código original dos autores foi refatorado em Python para garantir a legebilidade das computações e disponibilizado na Figura 23 e Figura 24.

Figura 23 – Computação das derivadas do FOAGDD em Python.

```python
# Biblioteca para manipulacao de vetores n-dimensionais
import numpy as np
# Biblioteca para operacao de convolucao
from scipy import signal

...
# Define o tensor que ira armazenar as derivadas de imagem geradas
derivadas = np.empty((numero_escalas,
                      numero_angulos,
                      largura_imagem,
                      altura_imagem),
                      dtype=np.float32)

# Computa e guarda a derivada de imagem em cada escala e angulo
for escala_index in range(numero_escalas):
  for angulo_index in range(numero_angulos):
    filtro = filtros[escala_index][angulo_index]
    derivada = signal.convolve2d(imagem, filtro, mode='same')
    derivadas[escala_index][angulo_index] = derivada
```

Fonte: Elaborada pelo autor.

Tendo em vista o alto fluxo de dados utilizados nesses componentes, o presente trabalho propõe acelerar suas computações pela implementação de versão paralela em GPU, usando a arquitetura CUDA da Nvídia.

3.2 Implementação do Método

Visando acelerar ainda mais o tempo de execução do extrator de cantos FOAGDD, uma abordagem pensada foi a adição de chamadas à GPU para paralelizar as seções de densa computação do código. Como a arquitetura de uma placa gráfica da NVIDIA possui à disposição centenas ou até milhares de *threads* para trabalhar com problemas densamente paralelizáveis, o suporte à esse recurso foi adicionado ao algoritmo.

Para essa abordagem, o código original do FOAGDD foi reescrito para a linguagem C++. Em seguida, para desenvolver ou utilizar códigos escritos na linguagem de programação CUDA com suporte à C++, foi necessário ligar os binários do CUDA SDK (*Software Development Kit*), com o compilador de C++. Ao fim desses procedimentos, tem-se todos os requerimentos necessários para criar a ponte de comunicação com a placa gráfica.

3.2.1 Biblioteca ArrayFire

O ArrayFire[1] de Yalamanchili et al. (2015), é uma biblioteca de alta performance com suporte para execuções na CPU, OpenCL e CUDA, disponível para as linguagens C, C++, Fortran

[1] https://arrayfire.com/. Acesso em: 12 dez. 2022.

Figura 24 – Computação das medidas de canto inicial do FOAGDD em Python.

```python
import numpy as np

...
# A janela de pixels para as medidas de canto
# tem tamanho N x N, porem na forma de um circulo.
# Sendo assim cria-se uma mascara para lidar
# com os pixels fora da area de interesse da janela.
# Mascara criada (0: pixels desativados, 1: pixels ativos):
#  0011100
#  0111110
#  1111111
#  1111111
#  1111111
#  0111110
#  0011100

tamanho_janela = 7 # Tamanho N da janela (N x N)

mascara = np.ones((patch_size,patch_size), dtype=bool)
mascara[0,:2] = False
mascara[0,-2:] = False
mascara[1,0] = False
mascara[1,-1] = False
mascara[-2,0] = False
mascara[-2,-1] = False
mascara[-1,:2] = False
mascara[-1,-2:] = False

# define o tensor que ira armazenar a medida
# de canto para cada pixel da imagem
medidas_canto = np.empty((altura, largura), dtype=np.float32)

# Itera pela coordenadas dos pixels da imagem
for (i,j) in np.ndindex(altura, largura):
    cima = i + tamanho_janela - 3
    baixo = i + tamanho_janela + 3
    esquerda = j + tamanho_janela - 3
    direita = j + tamanho_janela + 3

    # Recupere os pixels da janela na primeira
    # escala e em todos os angulos
    janela = derivadas[0, :, cima:baixo+1, esquerda:direita+1]

    # Nota: a janela acima possui dimensao
    # num_angulos x tamanho_janela x tamanho_janela

    # Remove os pixels fora da area de interessa da janela
    janela = janela[:, mask]

    # Nota: como uma mascara foi aplicada nas duas ultimas
    # dimensoes da janela, elas foram mescladas. Logo, agora
    # a janela possui dimensao
    # num_angulos x num_pixels_ativos_na_mascara

    # Multiplicacao de matrizes
    mat = janela @ janela.T
    # Computa e salva a medida de canto para o pixel atual
    medidas_canto[i, j] = np.linalg.det(mat) / (np.trace(mat) + eps)
```

Fonte: Elaborada pelo autor.

Figura 25 – Exemplo de uso da biblioteca arrayfire para convolução em lote.

```
...
#include <arrayfire.h> // inclui a biblioteca arrayfire

int main()
{
  /* Define os tamanhos de imagem e filtro,
  Assim como a quantia de filtros a serem utilizados */
  int tamanho_imagem = 512, tamanho_filtro = 7, numero_filtros=24;

  // Le a imagem e filtros, armazenando na CPU
  float *imagem_cpu = ...;
  float *filtros_cpu = ...;

  // Passa a imagem e filtros para a GPU
  af::array imagem_gpu(tamanho_imagem, tamanho_imagem, imagem_cpu);
  af::array filtros_gpu(tamanho_filtro, tamanho_filtro, numero_filtros
    , filtros_cpu);

  // Performa a convolucao em lote
  af::array resultado_gpu = af::convolve2(imagem_gpu, filtros_gpu);

  // Transfere o resultado da convolucao para a CPU
  float *resultado_cpu = resultado_gpu.host<float>();
}
```

Fonte: Elaborada pelo autor.

e Python. Ela contém um conjunto de implementações de funções relacionadas à computação em vetores nas áreas de manipulação de vetores, álgebra linear, visão computacional, estatística, entre outras.

Em particular para esse trabalho, o ArrayFire possui implementado em seus módulos a função `convolve2` que suporta diversas configurações de convoluções, por exemplo uma imagem e um filtro, múltiplas imagens e um filtro, múltiplos filtros e uma imagem , múltiplas imagens e múltiplos filtros. Visto que seguindo as configurações padrões do FOAGDD é necessário convolucionar 1 imagem com 24 filtros na etapa *Computação das derivadas* e essa operaçõa é custosa, a biblioteca ArrayFire foi acrescentada ao código para acelerar a computação das convoluções em lote. A implementação na Figura 25 elucida como isso pode ser feito utilizando a biblioteca mencionada.

3.2.2 CUDA kernels

Como o componente *Medida de cantos inicial* não representa nenhuma operação ou método padronizado em Computação para ser encontrado implementado em bibliotecas de alto desempenho, sua otimização na GPU é dependente da escrita de um CUDA kernel customizável que execute as mesmas operações, mas utilizando os recursos disponíveis na placa gráfica para acelerar as computações.

Figura 26 – Cálculo da configuração de *threads* utilizada pelo kernel do FOAGDD.

```
...
#define BLOCO_TAMANHO 32 // Numero de threads escolhido
...
int main()
{
  ...
  /* Nota: a arquitetura CUDA permite no maximo 1024 threads
  por bloco. Como nesse codigo sao utilizadas 2 dimensoes
  por bloco, tem-se 32 threads alocadas para a dimensao X
  e 32 para a dimensao Y, totalizando 1024 threads (32 * 32) */

  // Threads por bloco
  dim3 block_dim(BLOCO_TAMANHO,BLOCO_TAMANHO);
  // Blocos na grade
  dim3 grid_dim((largura_imagem+BLOCO_TAMANHO-1)/BLOCO_TAMANHO,
                (altura_imagem+BLOCO_TAMANHO-1)/BLOCO_TAMANHO);
  ...
}
```

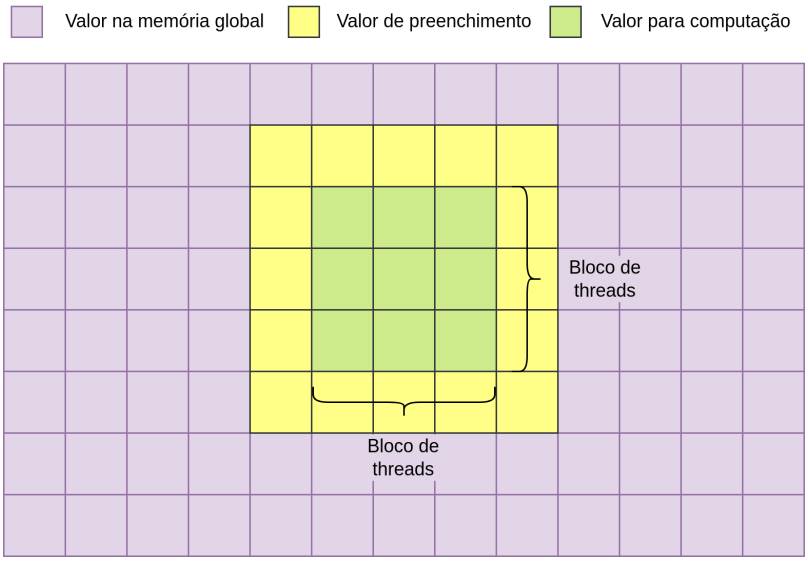
Fonte: Elaborada pelo autor.

Dito isso, a etapa mencionada foi reescrita na linguagem CUDA com suporte à C++ no formato de uma CUDA kernel, onde cada medida de canto é computada em uma *thread* da GPU. A configuração de *threads* definida é utilizando grade 2D e blocos 2D. Uma vez definido o número de *threads* em cada dimensão (X e Y) do bloco, o tamanho do grid é automaticamente calculado (Figura 26).

Como nessa operação existem muitos acessos aos valores das derivadas da imagem e o desempenho do código CUDA está intrinsicamente ligado à velocidade de transferência de dados no kernel, optou-se armazenar esses valores na memória compartilhada, uma vez que sua velocidade de acesso é rápida. Portanto, no início do kernel, todas as *threads* contidas no mesmo bloco copiam para a memória compartilhada os dados necessários provenientes da memória global, que são passados via parâmetros da função. A memória compartilhada em questão tem dimensão: `num_angulos` $\times$ `(BLOCK_SIZE + tamanho_janela - 1)` $\times$ `(BLOCK_SIZE + tamanho_janela - 1)`, portanto, algumas *threads* são ainda responsáveis por transferir valores de preenchimento da memória global para a compartilhada. A Figura 27 ilustra como os valores da memória global são guardados na memória compartilhada. Outrossim, o código desenvolvido na Figura 28 é uma simplificação de como essa cópia de valores é feita.

Com os valores necessários para a computação das medidas de canto salvos na memória compartilhada, o passo seguinte é a obtenção da matriz simétrica computada na linha 56 da Figura 24. Para isso, a multiplicação de matrizes em questão ($M \times M^T$) segue sua formulação computacional padrão, com a única diferença sendo que apenas o triângulo superior da matriz simétrica é gerado, para evitar computações extras, e, futuramente, o triângulo inferior é obtido

Figura 27 – Relação da memória global com a memória compartilhada no kernel de computação das medidas de canto inicial.



Fonte: Elaborada pelo autor

espelhando a matriz obtida na diagonal. Também, como nessa etapa é necessário levar em conta os pixels ativos `xy` da janela no momento da multiplicação, a coordenada desses pixels é pré-computada e salva na memória constante antes da chamada do *kernel*. O algoritmo na Figura 29 expõe a implementação desses passos.

Por fim, é necessário realizar a computação da medida de canto exposta na linha 58 da Figura 24. Sendo assim, os métodos `np.linalg.det` e `np.trace`, que calculam respectivamente o determinante e traço de uma matriz, foram implementados como *kernels* do *device* para atuarem na matriz obtida. Para fins de simplicidade, o código de formulação desses métodos não será mostrado, uma vez que possuem diversas implementações *online*. O algoritmo na Figura 30 implementa as computações restantes do kernel de medidas de canto inicial.

3.2.3 Considerações finais

Visando minimizar alocações de memória na placa gráfica e computações desnecessárias, o método FOAGDD foi englobado em uma classe para essa etapa. Como a utilização desse código é em aplicações que possuem imagens sequenciais e de tamanho fixo, essa alteração faz sentido de acordo com seu contexto. Logo, a alocação de memória na GPU e a etapa de *Computação dos filtros* são realizadas apenas na criação da classe. A Figura 31 procura clarificar essa adição.

Figura 28 – Simplificação da seção do kernel de computação das medidas de canto inicial que faz cópia da memória global para a memória compartilhada.

```
#define BLOCO_TAMANHO 32 // Numero de threads escolhido
__global__
void kernel(float* templates, // Derivadas da imagem
            size_t largura, // Largura da imagem
            size_t altura, // Altura da imagem
            size_t num_angulos, // Numero de angulos
            size_t tamanho_janela, // Tamanho N da janela (N x N), ...)
{
  // Index global das threads nas dimensoes X e Y
  int col_global = threadIdx.x + blockIdx.x * blockDim.x;
  int row_global = threadIdx.y + blockIdx.y * blockDim.y;

  // Cria a memoria compartilhada
  __shared__ float template_shr[num_angulos]
                               [BLOCO_TAMANHO + tamanho_janela - 1]
                               [BLOCO_TAMANHO + tamanho_janela - 1];

  // Copia 'num_angulos' blocos na memoria compartilhada
  for (size_t angulo_idx = 0;
       angulo_idx < num_angulos;
       angulo_idx++)
  // Itera pelos angulos
  {
    for (int row_offset = -BLOCO_TAMANHO;
             row_offset <= BLOCO_TAMANHO;
             row_offset += BLOCO_TAMANHO)
    // Itera pelos valores de preenchimento e computacao
    {
      for (int col_offset = -BLOCO_TAMANHO;
               col_offset <= BLOCO_TAMANHO;
               col_offset += BLOCO_TAMANHO)
      // Itera pelos valores de preenchimento e computacao
      {
        // Computa os indexes para a memoria compartilhada
        int curr_col = threadIdx.x + tamanho_janela/2 + col_offset;
        int curr_row = threadIdx.y + tamanho_janela/2 + row_offset;

        // Verifica se o index esta fora dos limites da memoria
        if(curr_col < 0 ||
           curr_col >= BLOCO_TAMANHO + tamanho_janela - 1 ||
           curr_row < 0 ||
           curr_row >= BLOCO_TAMANHO + tamanho_janela - 1)
        { continue; }

        float val = 0.f;
        // Verifica se o index esta fora dos limites da memoria global
        if(col_global + col_offset >= 0 &&
           col_global + col_offset < largura &&
           row_global + row_offset >= 0 &&
           row_global + row_offset < altura)
        {
          int pos_inicial = angulo_idx * altura;
          int altura_shift = pos_inicial + row_global + row_offset;
          int largura_shift = col_global + col_offset;
          // Acessa o index na memoria global
          val = *((templates + altura_shift * largura) + largura_shift);
        }

        // Preenche a memoria compartilhada
        template_shr[angulo_idx][curr_row][curr_col] = val;
      }
    }
  }
  // Sincroniza as threads do bloco antes de continuar
  __syncthreads(); ...
} ...
```

Fonte: Elaborada pelo autor.

Figura 29 – Simplificação da seção do kernel de computação das medidas de canto inicial que gera a matriz simétrica necessária para a computação das medidas de canto.

```
#define MAX 100
...
__constant__ uchar2 janela_pixels_validos[MAX];
void computa_janela_pixels_validos(size_t tamanho_janela)
{
  // Vetor temporario no host
  uchar2 h_janela_pixels_validos[MASK_MAX];
  size_t num_pixels_ativos = 0; // Contador

  // Computa as coordenadas xy dos pixels ativos da janela
  ...
  cudaMemcpyToSymbol(janela_pixels_validos,
                     h_janela_pixels_validos,
                     sizeof(uchar2)*num_pixels_ativos);
}

__global__
void kernel(...
            size_t num_angulos, // numero de angulos
            size_t tamanho_janela, // tamanho N da janela (N x N)
            ...)
{
  ...
  float template_simetrico[num_angulos][num_angulos];
  // Inicializa a matriz com 0s
  ...
  // Numero de pixels ativos da janela
  size_t num_mascara = tamanho_janela*tamanho_janela - 12;

  /* Multiplicacao de matrizes: A x A.T
  Como a matriz resultante tem simetria, as multiplicacoes
  sao feitas com relacao ao triangulo superior da matriz. O
  resultado disso sera espelhado para o triangulo inferior depois */

  for (size_t i = 0; i < num_angulos; i++)
  {
    for (size_t j = i; j < num_angulos; j++)
    {
      /* Percorra pela janela xy. As coordenadas de
      pixel validos da janela ja foram precomputadas */
      for (size_t k = 0; k < num_mascara; k++)
      {
        size_t curr_row = threadIdx.y + janela_pixels_validos[k].y;
        size_t curr_col = threadIdx.x + janela_pixels_validos[k].x;

        int val_a = im_template_shr[i][curr_row][curr_col];
        int val_b = im_template_shr[j][curr_row][curr_col];

        template_simetrico[i][j] += val_a * val_b;
      }
    }
  }

  // Espelha a matriz simetrica com relacao a sua diagonal
  for (size_t i = 0; i < num_angulos; i++)
  {
    for (size_t j = 0; j < i; j++)
    {
      template_simetrico[i][j] = template_simetrico[j][i];
    }
  } ...
} ...
```

Fonte: Elaborada pelo autor.

Figura 30 – Simplificação da seção do kernel de computação das medidas de canto inicial que calcula e guarda a medida de canto para um pixel.

```
__device__ float determinante(...){...}
__device__ float traco(...){...}

__global__
void kernel(...
            size_t largura, // Largura da imagem
            size_t altura, // Altura da imagem
            float* medidas_canto, // Medidas de canto (saida)
            float eps)
{
  // Index global das threads nas dimensoes X e Y
  int col_global = threadIdx.x + blockIdx.x * blockDim.x;
  int row_global = threadIdx.y + blockIdx.y * blockDim.y;
  ...
  float trc = traco(template_simetrico, ...);
  float det = determinante(template_simetrico, ...);

  float *medidas_canto_row = medidas_canto + row_global * largura;
  medidas_canto_row[col_global] = det / (trc + eps);
}
...
```

Fonte: Elaborada pelo autor.

Figura 31 – Abstração da classe elaborada para o FOAGDD.

```
class FOAGDD
{
  public:
  FOAGDD(configuracao_do_FOAGDD)
  {
    // Alocacao de memoria na placa grafica
    ...
    // Calculo dos filtros
    computa_filtros(configuracao_do_FOAGDD);
    ...
  }
  ~FOAGDD()
  {
    // Libera memoria alocada na placa grafica
    ...
  }
  void encontra_cantos(imagem, ...);
  {
    // Encontra os pontos de canto na imagem fornecida
    ...
  }

  private:
  void computa_filtros(...)
  {
    // Calcula os filtros de acordo com os parametros
    // do FOAGDD fornecidos no construtor da classe
    ...
  }
};
```

Fonte: Elaborada pelo autor.

4 Experimentos e Análise dos Resultados

Esse capítulo discute os experimentos realizados e seus respectivos resultados.

4.1 Ambiente experimental

4.1.1 Componentes de Hardware

A máquina utilizada no desenvolvimento dos experimentos a seguir contém as seguintes especificações de *hardware*:

- Processador (CPU): 2 vCPU Intel Xeon 2.199GHz
- Placa gráfica (GPU): NVIDIA Tesla T4, 16 GB VRAM
- Memória RAM: 12 GB

4.1.2 Componentes de Software

Os *softwares* utilizados no desenvolvimento dos experimentos a seguir contém as seguintes especificações:

- Sistema operacional: Ubuntu 18.04.6 LTS
- Compilador de C++: GCC 7.5.0
- Compilador de CUDA: NVCC 11.2, V11.2.152
- Versão de Python: 3.8.16
- Editor de código: Visual Studio Code

4.2 Código original

Apesar da implementação original de Zhang e Sun (2020) ter sido desenvolvida em MATLAB, o algoritmo disponível *online* pelos autores foi escrito em Python e está disponível no GitHub [1]. Além disso, para esses testes, uma alteração foi feita no código original para se alinhar com o método proposto pelos autores. No passo 5 do FOAGDD, um ponto é marcado como canto, caso sua medida de canto seja maior que um dado limite T_h em todas as escalas,

[1] https://github.com/RuyiDai19841002/Corner-Detection-Using-Multi-directional-Structure-Tensor-with-Multiple-Scales. Acesso em: 15 nov. 2022.

Figura 32 – Seção do código original do FOAGDD com divergência em relação ao método proposto no artigo (esquerda) e a respectiva alteração para se manter fiel ao método proposto no artigo (direita).

```
def foagdd(...):
  ...
  marked_img_two = np.zeros((rows,
                             cols))
  marked_img_three = np.zeros((rows,
                               cols))
  ...
  # Marque na matriz 'marked_img_two'
  # os pontos candidatos a canto
  # na 2a escala
  ...
  rr = np.array(
         marked_img_two.nonzero()).T
  ...
  # Itera pelos pontos candidatos
  # a canto obtidos na 2a escala
  for i in range(len(rr)):
    ...
    # Computa a medida de canto para
    # o pixel na 3a escala
    ...
    # Marque como candidato caso
    # a medida de canto seja maior
    # que o limite definido
    if KKKK > threshold:
      # Essa linha nao remove os pontos
      # que nao satisfazem a condicao de
      # medida de canto na 3a escala
      marked_img_two[rr[i, 0], \
                     rr[i, 1]] = 1
  rrr = np.array(
          marked_img_two.nonzero()).T
  return rrr
```

```
def foagdd(...):
  ...
  marked_img_two = np.zeros((rows,
                             cols))
  marked_img_three = np.zeros((rows,
                               cols))
  ...
  rr = np.array(
         marked_img_two.nonzero()).T
  ...
  for i in range(len(rr)):
    ...
    if KKKK > threshold:
      # Essa linha mantem o pixel como
      # candidato a canto caso atinja
      # a condicao de medida de canto
      # na 3a escala
      marked_img_three[rr[i, 0], \
                       rr[i, 1]] = 1
  rrr = np.array(
          marked_img_three.nonzero()).T
  return rrr
```

Fonte: Elaborada pelo autor

porém no algoritmo disponibilizado essa verificação é feita apenas até a segunda escala de um total de três. Portanto, a verificação da medida de canto na última escala foi adicionada ao algoritmo. Os algoritmos mostrados na Figura 32 relatam a mudança realizada.

4.3 Experimentos realizados

Para verificar a eficácia do algoritmo FOAGDD paralelizado para a GPU, duas métricas foram levadas em consideração:

- Diferença de detecções: essa métrica compara a quantia de pontos detectados na versão paralelizada em CUDA do extrator com relação à quantia de pontos identificados em sua versão original (sequencial, CPU). Essa métrica é calculada da seguinte maneira (Equação 4.1):

$$D_{dets} = N_p - N_s \tag{4.1}$$

onde N_s e N_p são, respectivamente, as quantias de pontos detectados no algoritmo sequencial e no paralelizado.

- *Speed-up*: essa métrica é resposável por comparar o ganho de velocidade obtido com a versão paralelizada em CUDA do extrator de pontos FOAGDD em relação à sua implementação original (sequencial, CPU). Essa métrica é calculada da seguinte maneira (Equação 4.2):

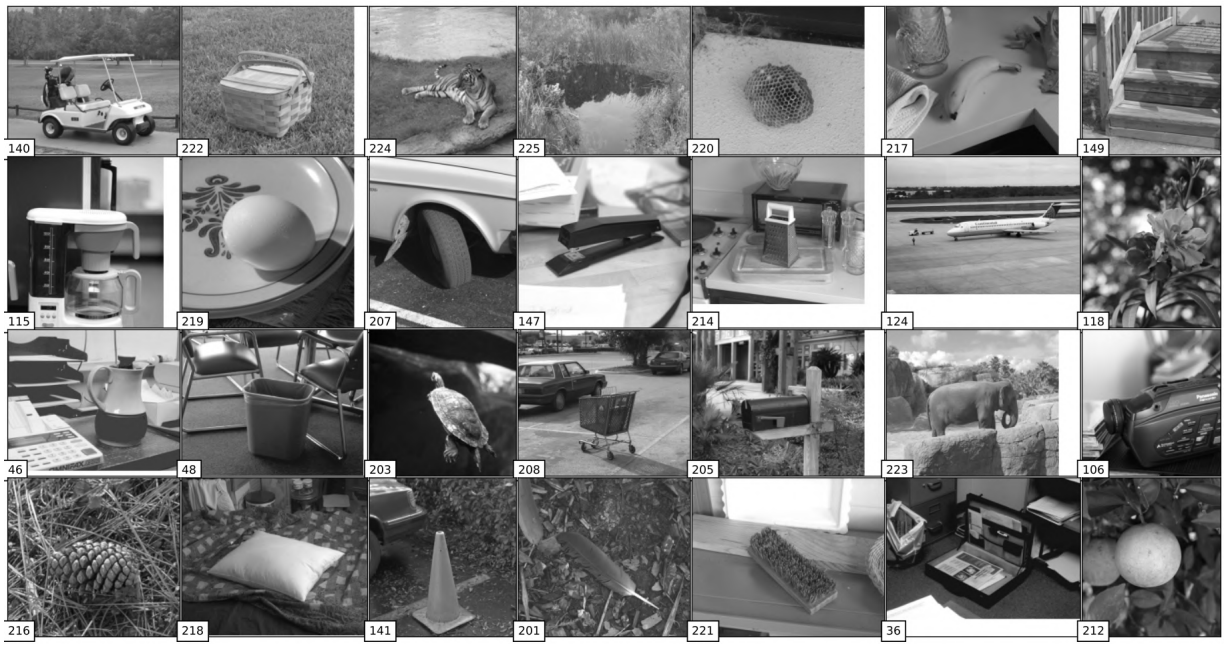
$$S = \frac{t_s}{t_p} \tag{4.2}$$

onde t_s e t_p são, respectivamente, os tempos de execução do algoritmo sequencial e do paralelizado.

4.3.1 Diferença de detecções

Para essa etapa, a relação de pontos detectados entre a versão sequencial (CPU) e em paralelo (GPU) do método foi feita com 28 imagens (Bowyer, Kranenburg e Dougherty (1999)) do dataset da USF (*University of South Florida*), utilizadas também na seção de avaliação do FOAGDD no artigo original. Essas imagens e seus respectivos nomes podem ser vistos na Figura 33.

Figura 33 – Imagens utilizadas para avaliação das detecções entre o método sequencial e o paralelo em CUDA. O nome das figuras está nas caixas brancas abaixo de cada imagem.

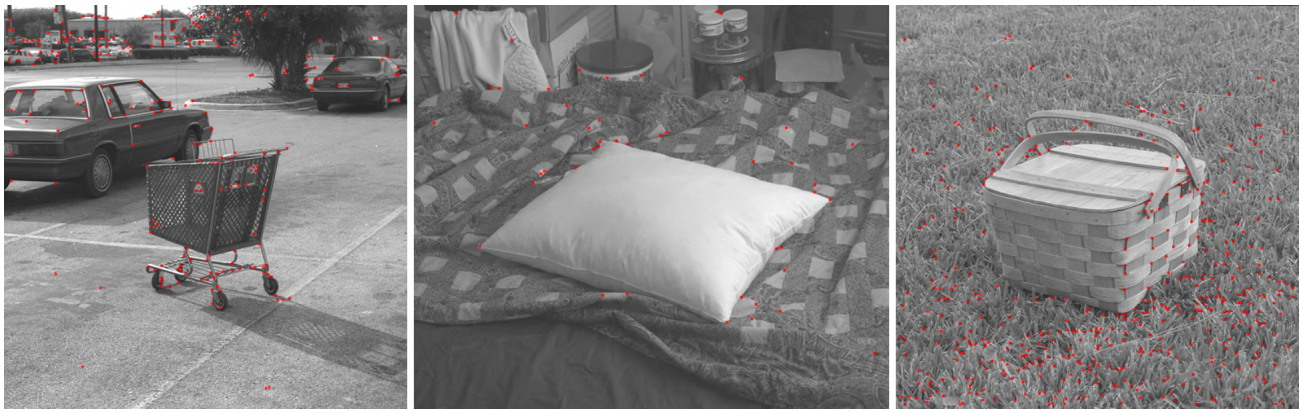


Fonte: Bowyer, Kranenburg e Dougherty (1999)

4.3.1.1 Resultados

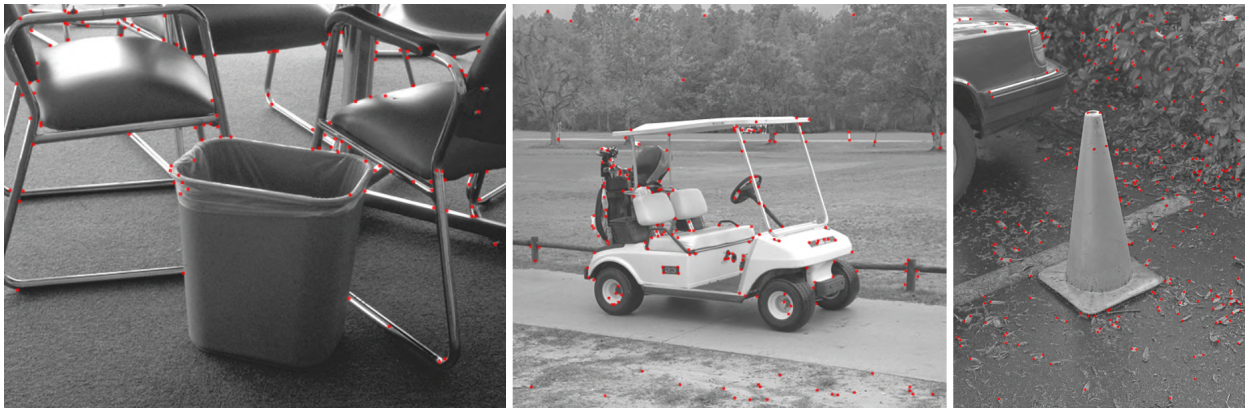
A Tabela 1 compara os pontos detectados pela implementação original do FOAGDD e sua respectiva versão em paralelo na GPU. Como é possível notar, a taxa de acertos é muito próxima entre as duas implementações. A última coluna da Tabela 1 exibe a diferença entre a quantidade de pontos detectados pela implementação paralelizada em comparação com a implementação original. Portanto, valores negativos indicam que a implementação paralelizada encontrou menos pontos do que a implementação original. A pequena variação entre a quantia de cantos detectados pode estar ligada à precisão numérica dos cálculos, uma vez que são feitos utilizando dados do tipo `float`. Sendo assim, pode-se entender que ambas implementações produzem resultados muito próximos. Na Figura 34 e 35 são mostradas, em vermelho, as detecções obtidas pelo código paralelizado em 6 imagens do dataset utilizado.

Figura 34 – Detecções de cantos pelo algoritmo paralelizado em algumas amostras do dataset da USF (nome das imagens da esquerda para a direita: 208, 218, 222).



Fonte: Elaborada pelo autor.

Figura 35 – Detecções de cantos pelo algoritmo paralelizado em algumas amostras do dataset da USF (nome das imagens da esquerda para a direita: 48, 140, 141).



Fonte: Elaborada pelo autor.

Tabela 1 – Comparação da quantia de pontos detectados entre o método sequencial e o paralelo em CUDA.

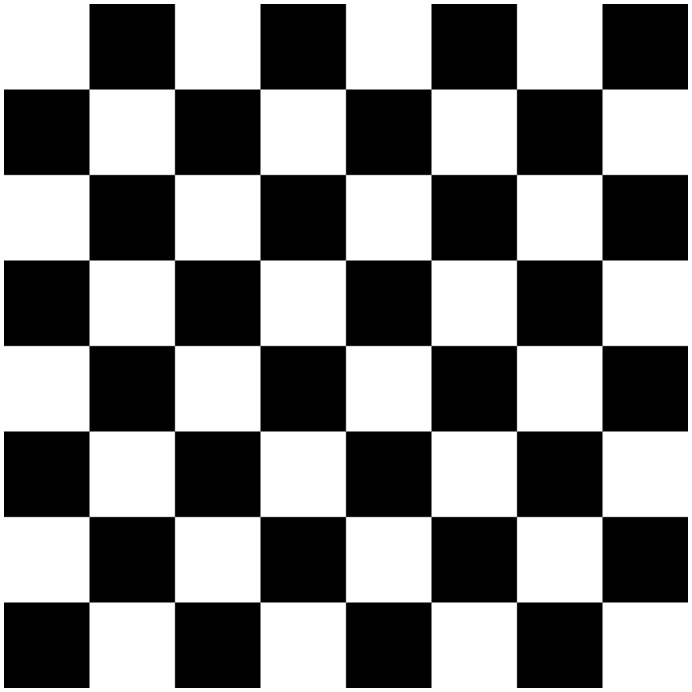
Nome imagem	Pontos detectados (Original)	Pontos detectados (Paralelo)	D_{dets}
36	157	157	0
46	155	155	0
48	125	125	0
106	148	149	1
115	127	126	-1
118	194	194	0
124	131	130	-1
140	180	181	1
141	258	263	5
147	30	30	0
149	222	222	0
201	382	382	0
203	102	102	0
205	540	540	0
207	76	76	0
208	240	240	0
212	12	12	0
214	89	89	0
216	1302	1302	0
217	24	24	0
218	46	46	0
219	50	50	0
220	157	157	0
221	107	108	1
222	439	439	0
223	279	279	0
224	97	97	0
225	407	407	0

Fonte: Elaborada pelo autor.

4.3.2 Speed-up

Visando manter um ambiente de teste mais consistente possível, foram produzidas uma série de imagens similares a um tabuleiro de xadrez (Figura 36) de resoluções variadas: 512 x 512, 640 x 480, 1280 x 720, 1600 x 1200 e 1920 x 1080. Essas imagens possuem a mesma quantia de quadrados e seus redimensionamentos foram feitos utilizando interpolação por vizinho mais próximo (*Nearest-neighbor interpolation*), para não haver nenhuma diferença na cor dos pixels entre elas. Como o contexto dessas imagens é o mesmo, o objetivo do teste se especializa em entender como a velocidade da paralelização proposta se comporta com aumento de área sucessivo na imagem de entrada

Figura 36 – Exemplo de imagem similar a um tabuleiro de xadrez, utilizada para averiguar as detecções de cantos.



Fonte: Elaborada pelo autor

4.3.2.1 Ferramenta para inferência do tempo

Para determinar o tempo médio gasto em cada código usado no desenvolvimento do atual trabalho, uma classe para a inferência de tempo foi criada, com a finalidade de padronizar essas medidas. Sendo assim, cada algoritmo é executado N vezes e o tempo levado em cada execução é acumulado em uma variável. Por fim, esse somatório é divido pelo número N de chamadas e tem-se o tempo médio gasto. Para evitar a latência causada pela primeira execução do programa, sempre a medida inicial é descartada.

4.3.2.2 Resultados

Na Tabela 2 é comparado o tempo de execução do FOAGDD entre sua implementação original e a paralelizada em CUDA em cada etapa do algoritmo ao variar o tamanho da imagem de entrada. Consoante com a tabela gerada, foi produzido o gráfico da Figura 37, que evidencia a métrica de *Speed-up* de acordo com o tamanho da imagem.

Desses resultados, pode-se retirar as seguintes conclusões:

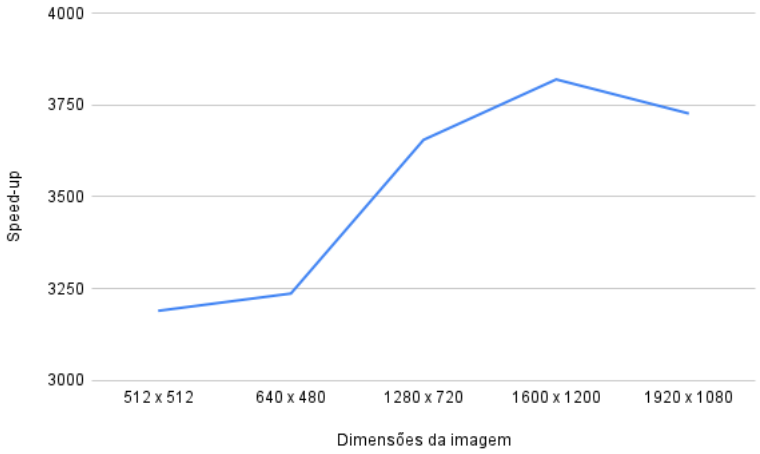
- Para imagens de tamanho até 640 x 480 o extrator FOAGDD otimizado para utilizar os recursos de computação da GPU é suficiente para prover inferências em tempo real.
- Para imagens de resolução igual ou acima de HD (*High Definition*), 1280 x 720, o algoritmo otimizado não é suficiente para providenciar extrações de canto em tempo real.

Tabela 2 – Comparação do tempo em milisegundos em cada componente do FOAGDD entre a implementação original e a otimização em GPU, variando o tamanho da imagem de entrada.

		Preparação da imagem	Computação dos filtros	Computação das derivadas	Medida de cantos inicial	Supressão não-máxima	Medida de cantos restante	Total	*Speed Up*
512 x 512	Original	0,50	748,24	15935,97	49299,69	5,36	44,38	66034,14	3190
	Otimizado (GPU)	0,00	-	10,00	9,60	1,10	0,00	20,70	
640 x 480	Original	0,53	740,11	18278,18	58616,59	6,31	47,91	77689,62	3237
	Otimizado (GPU)	0,00	-	10,10	11,90	2,00	0,00	24,00	
1280 x 720	Original	1,54	784,27	52975,19	175353,34	16,43	53,23	229184,00	3655
	Otimizado (GPU)	1,80	-	24,50	31,40	5,00	0,00	62,70	
1600 x 1200	Original	2,91	785,72	110435,17	367280,59	33,72	66,81	478604,92	3820
	Otimizado (GPU)	3,00	-	48,70	62,60	11,00	0,00	125,30	
1920 x 1080	Original	2,82	784,24	116001,51	375424,55	35,39	70,30	492318,80	3727
	Otimizado (GPU)	3,20	-	50,10	66,60	12,20	0,00	132,10	

Fonte: Elaborada pelo autor.

Figura 37 – *Speed-up* de acordo com a resolução da imagem.



Fonte: Elaborada pelo autor

5 Conclusão

Aspirando melhorar o tempo de execução do extrator de cantos FOAGDD para funcionar em tempo real, foi proposto aproveitar os recursos computacionais altamente paralelizáveis de uma GPU para acelerar as computações do algoritmo. Sendo assim, o código original do método foi reescrito na linguagem CUDA C++, que possibilita uma ponte de comunicação entre a CPU e a arquitetura CUDA disponível nas placas de vídeo da NVIDIA. A paralelização adicionada procura acelerar duas áreas de densa computação do algoritmo:

1. **Computação das derivadas:** Aplica operações de convolução em lote, entre a imagem de entrada e múltiplos filtros 2D, para produzir derivadas da imagem. A solução de paralelização dessas computações veio com a adição da biblioteca de alto desempenho ArrayFire, que implementa diversos algoritmos relacionados á visão computacional, incluindo convoluções em lote, de maneira otimizada e com suporte à arquitetura CUDA.

2. **Medidas de canto inicial:** Computa a medida de canto para cada pixel/valor das derivadas da imagem em múltiplas escalas. A solução de paralelização encontrada para essa etapa foi o desenvolvimento de um CUDA kernel que utiliza as diversas threads presentes na GPU para trabalhar em conjuntos nas computações.

Em seguida, para testar a eficiência das abordagens propostas, foram elaboradas duas métricas, a primeira compara a quantia de pontos detectados entre as versões: sequencial (CPU) e paralela (GPU) do FOAGDD e a segunda calcula o *Speed-up* obtido com a paralelização em CUDA. Com isso, pode-se concluir que a implementação do método com suporte à GPU produz detecções de pontos muito próximas das obtidas com o código original, que é serial e executa na CPU. Também, foi possível entender que resoluções de imagem inferiores a 1280×720(HD) são suficientes para serem processadas em tempo real pelo código paralelizado, atingindo um *Speed-up* de 3190 (66,03 segundos $\rightarrow$ 20,70 milisegundos) utilizando como base uma imagem 512 $\times$ 512 pixels. Porém, figuras com resoluções iguais ou superiores a HD ainda sofrem nesse quesito. O algoritmo paralelo desenvolvido nessa pesquisa pode ser acessado a partir do seguinte link: https://github.com/GustavoStahl/Corner-Extraction-FOAGDD-CUDA.

Por fim, como sugestão para trabalhos futuros, pode-se tentar medir o desempenho do código paralelizado em máquinas com placas gráficas de poder computacional superior à utilizada no presente trabalho. Outrossim, pode-se investigar possíveis melhorias dentro das operações de paralelização propostas, como escrever um kernel próprio para as convoluções em lote ou procurar operações que acelerem o kernel que computa as medidas de canto inicial. Ainda mais, é possível avaliar a abordagem proposta em um conjunto de imagens maior, para generalizar seu cenário de funcionamento.

Referências

ALCANTARILLA, P. F.; BARTOLI, A.; DAVISON, A. J. Kaze features. In: SPRINGER. *European conference on computer vision*. [S.l.], 2012. p. 214–227.

ARSENEAU, S. *Junction analysis: representing junctions through asymmetric tensor diffusions*. [S.l.]: VDM Publishing, 2008.

AWRANGJEB, M.; LU, G. Robust image corner detection based on the chord-to-point distance accumulation technique. *IEEE transactions on multimedia*, IEEE, v. 10, n. 6, p. 1059–1072, 2008.

BAO, W.; WANG, W.; XU, Y.; GUO, Y.; HONG, S.; ZHANG, X. Instereo2k: A large real dataset for stereo matching in indoor scenes. *Science China Information Sciences*, Springer, v. 63, n. 11, p. 1–11, 2020.

BAY, H.; TUYTELAARS, T.; GOOL, L. V. Surf: Speeded up robust features. In: SPRINGER. *European conference on computer vision*. [S.l.], 2006. p. 404–417.

BOWYER, K.; KRANENBURG, C.; DOUGHERTY, S. Edge detector evaluation using empirical roc curves. In: *Proceedings. 1999 IEEE Computer Society Conference on Computer Vision and Pattern Recognition (Cat. No PR00149)*. [S.l.: s.n.], 1999. v. 1, p. 354–359 Vol. 1.

BROWN, M.; LOWE, D. G. Automatic panoramic image stitching using invariant features. *International journal of computer vision*, Springer, v. 74, n. 1, p. 59–73, 2007.

BURGER, W.; BURGE, M. J. *Principles of digital image processing: core algorithms*. [S.l.]: Springer Science & Business Media, 2010.

CHENG, J.; GROSSMAN, M.; MCKERCHER, T. *Professional CUDA C programming*. [S.l.]: John Wiley & Sons, 2014. (1).

DERPANIS, K. G. The harris corner detector. *York University*, v. 2, p. 1–2, 2004.

DETONE, D.; MALISIEWICZ, T.; RABINOVICH, A. Superpoint: Self-supervised interest point detection and description. In: *Proceedings of the IEEE conference on computer vision and pattern recognition workshops*. [S.l.: s.n.], 2018. p. 224–236.

DUSMANU, M.; ROCCO, I.; PAJDLA, T.; POLLEFEYS, M.; SIVIC, J.; TORII, A.; SATTLER, T. D2-net: A trainable cnn for joint description and detection of local features. In: *Proceedings of the ieee/cvf conference on computer vision and pattern recognition*. [S.l.: s.n.], 2019. p. 8092–8101.

FORSYTH, D. A.; PONCE, J. *Computer vision: a modern approach*. [S.l.]: prentice hall professional technical reference, 2002.

GHORPADE, J.; PARANDE, J.; KULKARNI, M.; BAWASKAR, A. Gpgpu processing in cuda architecture. *arXiv preprint arXiv:1202.4347*, 2012.

GUPTA, P. *CUDA Refresher: The CUDA Programming Model*. NVIDIA, 2020. Disponível em: <https://developer.nvidia.com/blog/cuda-refresher-cuda-programming-model/>. Acesso em: 06 jan. 2022.

HARRIS, C.; STEPHENS, M. et al. A combined corner and edge detector. In: CITESEER. *Alvey vision conference*. [S.l.], 1988. v. 15, n. 50, p. 10–5244.

HE, X.; YUNG, N. H. C. Corner detector based on global and local curvature properties. *Optical engineering*, SPIE, v. 47, n. 5, p. 057008, 2008.

HEFFERON, J. Linear algebra third edition. 2018.

JING, J.; GAO, T.; ZHANG, W.; GAO, Y.; SUN, C. Image feature information extraction for interest point detection: A comprehensive review. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, IEEE, 2022.

KITANI, K. *Harris Corners*. 2017. Apresentação de Slides. Disponível em: <https://www.cs.cmu.edu/~16385/s17/Slides/6.2_Harris_Corner_Detector.pdf>. Acesso em: 14 dez. 2022.

KLIPPENSTEIN, J.; ZHANG, H. Quantitative evaluation of feature extractors for visual slam. In: IEEE. *Fourth Canadian Conference on Computer and Robot Vision (CRV'07)*. [S.l.], 2007. p. 157–164.

KRINGS, E. *Understanding Common FPS Values: The Advanced Guide to Video Frame Rates [2022 Update]*. Dacast, 2022. Disponível em: <https://www.dacast.com/blog/frame-rate-fps/>. Acesso em: 31 dez. 2022.

LANGE, K.; KIDD, C. *IT Benchmarking Explained: How To Assess Your IT Efforts*. BMC, 2021. Disponível em: <https://www.bmc.com/blogs/it-benchmarking-metrics/#:~:text=Benchmarking%20is%20a%20formal%20way,It%20is%20primarily%20data%2Ddriven.> Acesso em: 24 dez. 2022.

LOWE, D. G. Distinctive image features from scale-invariant keypoints. *International journal of computer vision*, Springer, v. 60, n. 2, p. 91–110, 2004.

MEDIONI, G.; FRANÇOIS, A. R.; SIDDIQUI, M.; KIM, K.; YOON, H. Robust real-time vision for a personal service robot. *Computer Vision and Image Understanding*, Elsevier, v. 108, n. 1-2, p. 196–203, 2007.

MIKOLAJCZYK, K.; SCHMID, C. Scale & affine invariant interest point detectors. *International journal of computer vision*, Springer, v. 60, n. 1, p. 63–86, 2004.

OCHOA, B. *Edge Detection and Corner Detection*. 2016. Apresentação de Slides. Disponível em: <https://cseweb.ucsd.edu/classes/sp16/cse152-a/lec7.pdf>. Acesso em: 26 dez. 2022.

ONO, Y.; TRULLS, E.; FUA, P.; YI, K. M. Lf-net: Learning local features from images. *Advances in neural information processing systems*, v. 31, 2018.

QUANDT, M.; KNOKE, B.; GORLDT, C.; FREITAG, M.; THOBEN, K.-D. General requirements for industrial augmented reality applications. *Procedia Cirp*, Elsevier, v. 72, p. 1130–1135, 2018.

REY-OTERO, I.; DELBRACIO, M.; MOREL, J.-M. Comparing feature detectors: A bias in the repeatability criteria. In: IEEE. *2015 IEEE International Conference on Image Processing (ICIP)*. [S.l.], 2015. p. 3024–3028.

ROSTEN, E.; DRUMMOND, T. Fusing points and lines for high performance tracking. In: IEEE. *Tenth IEEE International Conference on Computer Vision (ICCV'05) Volume 1*. [S.l.], 2005. v. 2, p. 1508–1515.

SÁNCHEZ, J.; MONZÓN, N.; NUEZ, A. S. D. L. An analysis and implementation of the harris corner detector. *Image Processing On Line*, 2018.

SHUI, P.-L.; ZHANG, W.-C. Corner detection and classification using anisotropic directional derivative representations. *IEEE Transactions on Image Processing*, IEEE, v. 22, n. 8, p. 3204–3218, 2013.

STORTI, D.; YURTOGLU, M. *CUDA for engineers: an introduction to high-performance parallel computing*. [S.l.]: Addison-Wesley Professional, 2015.

SVELEBA, S.; KATERYNCHUK, I.; KARPA, I.; KUNYO, I.; UGRYN, S.; UGRYN, V. The real time face recognition. In: *2019 3rd International Conference on Advanced Information and Communications Technologies (AICT)*. [S.l.: s.n.], 2019. p. 294–297.

SZELISKI, R. *Computer vision: algorithms and applications*. [S.l.]: Springer Nature, 2022.

TANG, P.; WANG, Z.; QI, N.; ZHU, Q. A fast feature extraction process for visual slam. In: IEEE. *2019 IEEE 3rd Information Technology, Networking, Electronic and Automation Control Conference (ITNEC)*. [S.l.], 2019. p. 959–963.

TYAGI, D. *Introduction to feature detection and matching*. Medium, 2019. Disponível em: <https://medium.com/data-breach/introduction-to-feature-detection-and-matching>. Acesso em: 1 dez. 2022.

ULRICH, M.; STEGER, C.; BAUMGARTNER, A. Real-time object recognition using a modified generalized hough transform. *Pattern Recognition*, Elsevier, v. 36, n. 11, p. 2557–2570, 2003.

WALT, S. Van der; SCHÖNBERGER, J. L.; NUNEZ-IGLESIAS, J.; BOULOGNE, F.; WARNER, J. D.; YAGER, N.; GOUILLART, E.; YU, T. scikit-image: image processing in python. *PeerJ*, PeerJ Inc., v. 2, p. e453, 2014.

Wikipedia contributors. *Image gradient — Wikipedia, The Free Encyclopedia*. 2022. [Online; accessed 26-December-2022]. Disponível em: <https://en.wikipedia.org/w/index.php?title=Image_gradient&oldid=1119649092>.

XIA, G.-S.; DELON, J.; GOUSSEAU, Y. Accurate junction detection and characterization in natural images. *International journal of computer vision*, Springer, v. 106, n. 1, p. 31–56, 2014.

YALAMANCHILI, P.; ARSHAD, U.; MOHAMMED, Z.; GARIGIPATI, P.; ENTSCHEV, P.; KLOPPENBORG, B.; MALCOLM, J.; MELONAKOS, J. *ArrayFire - A high performance software library for parallel computing with an easy-to-use API*. Atlanta: ArrayFire, 2015. Disponível em: <https://github.com/arrayfire/arrayfire>.

YAQOOB, I.; KHAN, L. U.; KAZMI, S. M. A.; IMRAN, M.; GUIZANI, N.; HONG, C. S. Autonomous driving cars in smart cities: Recent advances, requirements, and challenges. *IEEE Network*, v. 34, n. 1, p. 174–181, 2020.

YI, K. M.; TRULLS, E.; LEPETIT, V.; FUA, P. Lift: Learned invariant feature transform. In: SPRINGER. *European conference on computer vision*. [S.l.], 2016. p. 467–483.

ZHANG, W.; SUN, C. Corner detection using second-order generalized gaussian directional derivative representations. *IEEE transactions on pattern analysis and machine intelligence*, IEEE, v. 43, n. 4, p. 1213–1224, 2019.

ZHANG, W.; SUN, C. Corner detection using multi-directional structure tensor with multiple scales. *International Journal of Computer Vision*, Springer, v. 128, n. 2, p. 438–459, 2020.

ZHANG, W.; SUN, C.; BRECKON, T.; ALSHAMMARI, N. Discrete curvature representations for noise robust image corner detection. *IEEE Transactions on Image Processing*, IEEE, v. 28, n. 9, p. 4444–4459, 2019.

ZHANG, X.; WANG, H.; SMITH, A. W.; LING, X.; LOVELL, B. C.; YANG, D. Corner detection based on gradient correlation matrices of planar curves. *Pattern recognition*, Elsevier, v. 43, n. 4, p. 1207–1223, 2010.

ZHANG, X.; YU, F. X.; KARAMAN, S.; CHANG, S.-F. Learning discriminative and transformation covariant local feature detectors. In: *Proceedings of the IEEE conference on computer vision and pattern recognition*. [S.l.: s.n.], 2017. p. 6818–6826.